CANADIAN THESES ON MICROFICHE

I.S.B.N.

THESES CANADIENNES SUR MICROFICHE

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us a poor photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de mauvaise qualité.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

NL-339 (r. 82/08)

Canada

National Library of Canada | Bibliothèque nationale du Canada

Canadian Theses Division | Division des thèses canadiennes

Ottawa, Canada
K1A 0N4

0-315-06091-3

54011

PERMISSION TO MICROFILM — AUTORISATION DE MICROFILMER

• Please print or type — Écrire en lettres moulées ou dactylographier

Full Name of Author — Nom complet de l'auteur

MARIUS OLAFSSON

Date of Birth — Date de naissance	Country of Birth — Lieu de naissance
JUNE 05 1955	ICELAND

Permanent Address — Résidence fixe

FLYÐRUGRANDI 4 B-1
102 REYKJAVIK ICELAND

Title of Thesis — Titre de la thèse

THE QM-G: A G-ORIENTED INSTRUCTION SET ARCHITECTURE FOR THE NANODATA QM-1

University — Université

UNIVERITY OF ALBERTA

Degree for which thesis was presented — Grade pour lequel cette thèse fut présentée

MASTER OF SCIENCE

Year this degree conferred — Année d'obtention de ce grade	Name of Supervisor — Nom du directeur de thèse
1981	SUBRATA DASGUPTA

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.

Date	Signature
AUGUST 31, 1981	Marius Olafsson

NL-91 (4/77)

THE UNIVERSITY OF ALBERTA

The QM-C: A C-oriented Instruction Set Architecture for the Nanodata QM-1

by

© Marius Olafsson

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF Master of Science

Department of Computing Science

EDMONTON, ALBERTA

Fall 1981

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR Marius Olafsson

TITLE OF THESIS The QM-C: A C-oriented Instruction Set Architecture for the Nanodata QM-1

DEGREE FOR WHICH THESIS WAS PRESENTED Master of Science

YEAR THIS DEGREE GRANTED Fall 1981

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(SIGNED) Marius Olafsson

PERMANENT ADDRESS:

Flydrugrandi 4, B-1

102 Reykjavík

Iceland.

DATED August 27. 1981

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled The QM-C: A C-oriented Instruction Set Architecture for the Nanodata QM-1 submitted by Marius Olafsson in partial fulfilment of the requirements for the degree of Master of Science.

.....

Supervisor

.....

.....

.....

Date AUGUST 27, 1981

To Helga

ABSTRACT

With the advent of user-microprogrammable computers, it became possible to design experimental systems whose architecture is oriented toward efficient execution of high level languages. This kind of experimentation is necessary if we are to assess the merits and drawbacks of possible architectural solutions to the disparity between modern high level languages and the machines on which they execute.

This work presents the design of the QM-C, an architecture directed toward the execution of the programming language C on the Nanodata QM-1. For microprogrammed architectures to be practical, their design must be a "compromise" between the architectural features dictated by their intended application and the technological limitations imposed by the host machine. In order to achieve this compromise, a rigorous, top-down method is applied in the design the QM-C. The design is preceded by an extensive analysis of the usage of the C language. The results and applications of this study to the design process are discussed.

The QM-C architecture is formally described using a newly developed architectural description language (S*A). This description is then transformed into a representation in the high level microprogramming language S*(QM-1). An evaluation of these two languages is presented and conclusions drawn about their usage and applicability.

Preliminary evaluation of the QM-C architecture indicates its superiority over the MULTI instruction set, traditionally used to program the QM-1.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dr. Subrata Dasgupta, for his constant encouragement and constructive criticism throughout the course of this work. I am grateful to Steven Sutphen for introducing the subject to me and for his many helpful suggestions.

I would also like to thank the other members of the 'QM-1 Group', namely Alynn Klassen and Doug Rideout, for the numerous discussions we had during my stay here. These discussions contributed greately to this work.

I am indebted to Mary Angela Papalaskaris for her careful reading of the earlier drafts of this thesis and valuable criticism. Also, I am grateful to Dr. Rick Heuft, Lisa Higham and Leo Hartman for their comments on this work.

Finally, special thanks to my wife Helga, to whom this work is dedicated.

Table of Contents

Chapter		Page
1.	Introduction	1
	1.1 Related Work	6
2.	Methodology of the Design	10
	2.1 Top-Down vs. Bottom-Up Design	11
	2.2 Design Compromise	12
3.	Empirical Study of C Programs	14
	3.1 The Sample	14
	3.2 Compiler Instrumentation	15
	3.2.1 Sources of Error	17
	3.3 Operators	18
	3.4 Statements	20
	3.5 Data Types and Storage Classes	29
	3.6 Size Characteristics	32
	3.7 Summary	38
4.	The QM-C Execution Architecture	39
	4.1 Design Alternatives	40
	4.2 Storage Organization	43
	4.3 Addressing Modes	46
	4.4 Type Mapping	47
	4.4.1 Structures, Strings, Arrays and Pointers	49
	4.5 Instruction Formats	50
	4.6 Operator Selection	53
	4.6.1 Reduced vs Complex Instruction Set	54
	4.7 Instruction Set Overview	55
	4.7.1 Functional Instructions (F-type)	55
	4.7.2 Procedural Instructions (P-type)	56
	4.7.3 Move Instructions (M-type)	58
5.	On Architectural Description and Implementation	60
	5.1 The S*A Architectural Description Language	61

5.2 The S*(QM-1) Microprogramming Language 64
5.3 Transformation from Description to Implementation 65
6. Conclusions 70
6.1 Future Work 71
Bibliography 74
Appendix I 82
Appendix II 86
Appendix III 92
Appendix IV 105

List of Figures

Figure		Page
1.	Complexity vs Design Direction	3
2.	The Development of a language-directed architecture	4
3.	Usage of the QM-C	5
4.	QM-C's Design Process	11
5.	QM-1's Emulator Environment	40
6.	QM-C/UNIX Environment	40
7.	QM-C's Register File	44
8.	QM-C's Stack Mechanism	45
9.	Instruction Decode Mechanism	51
10.	S*A Description	67
11.	S*(QM-1) Program Portion	68
12.	C Example Program	88
13.	QM-C Code Example	89
14.	MULTI Implementation	91

List of Tables

Table		Page
1.	Arithmetic Operators	18
2.	Relational and Logical Operators	19
3.	Assignment Operators	21
4.	Statement Types	22
5.	Average Number of Statements Between Procedure Calls	23
6.	Switch Statements	24
7.	Assignment Types	26
8.	Arithmetic Expression Types	27
9.	Relational and Logical Expression Types	28
10.	Distribution of Constant Magnitudes	29
11.	Basic Types	30
12.	Typing Complexity	30
13.	Storage Classes	31
14.	Reference Density	33
14a.	Reference to Complex Types	33
15.	Stack Offset (arguments)	35
15a.	Stack Offset (automatics+temp.)	35
16.	Average Number of Parameters per Procedure called	36
16a.	Number of Parameters per Procedure Defined	36

17. Distribution of Automatic Variables Defined 37

18. Distribution of Register Variables Defined 37

19. Instruction Formats 53

20. Instruction Set Summary 58

21. Language Overview 62

Chapter 1

Introduction

Problem-solving using digital computers inevitably involves some kind of mapping from the initial problem representation to a representation understood and executable by the machine. This *machine language* is far removed from the original problem as conceived by the person using the machine and consequently this mapping becomes complex and error-prone. To remedy this situation *high-level languages* were introduced and programs provided to translate these languages to the underlying machine representation. Although these programs - *compilers* - greatly eased the burden on the user, they were never quite successful in translating the high level language problem representation *efficiently* to an equivalent machine language representation. The primary reason for this inability was the complexity of the transformation caused by the disparity between the semantic characteristics of the high-level language on one hand, and the machine language on the other.

Because high-level languages continue to evolve as people learn more about programming methodology in general, the translation complexity increases. It is now apparent that the machine language must somehow be made to absorb some of this complexity. The investigation into how the machine's logical characteristics (its *architecture*) may be changed to accommodate some of the problem/machine mapping is the overall context into which this thesis fits.

Reducing (or distributing) the complexity of the aforementioned translation can be accomplished either directly by reconfiguring the machine's architecture for each problem [saun79], or indirectly by *orienting* the architecture toward the execution of the high-level language in which the problem is expressed. Thus, the design goal of the architect of a language-oriented machine is the reduction of the language/machine mapping complexity by moving the machine language closer to the high level language. This work deals with the indirect approach.

With the advent of user-microprogrammable machines[1] it became possible to design and experiment with language oriented architectures realized by microprogramming on actual machines. This kind of experimentation is essential if one is to assess the merits and drawbacks of possible architectural solutions to the disparity between the high level language and the machine architecture on which it executes. (This disparity has been termed *The Semantic Gap* by Myers [myer78] and others.)

Architectures oriented toward 'better' execution of high-level languages exist basically in three forms [chu75,myer78]:

1. **High-Level Language Machines** (HLLM's).
   Here, the architecture is directed toward the language such that the high-level language is the *assembly* language of the system. There is a one-to-one correspondence between the language constructs and the machine operators. The programs that map the high-level language to the machine representation perform none of the traditional compiler functions. Some HLLM's interpret the high-level language directly with no translation. For microprogrammable machines the complexity of the problem/machine mapping has been wholly transferred into the microprogrammed interpreter for the language.
2. **DEL's** (Directly Executable Languages [hove78]).
   Here, the high-level language is compiled into an intermediate representation (DEL). The machine interprets this representation directly in microcode. The compilation is more complicated than for HLLM's but high interpreter complexity is still needed to achieve the primary design goals of the DEL's: one-to-one correspondence between the operators and data objects of the high level language and those of the intermediate representation. Thus, the design of a DEL is completely dictated by the high-level language.
3. **Language-Directed Machines**. In designing language directed architectures an attempt is made to distribute the complexity of the problem-to-machine mapping evenly between the compilation of the high-level language and the microprograms that realize the architecture. The resulting architecture is a compromise between the

---

[1]Examples of these machines include the Nanodata QM-1 [nano79], the QA-1 [hagi80], ,the BBN's Microprogrammable Building Block [kral80], and the Burroughs B1700/B1800 series [orga78]

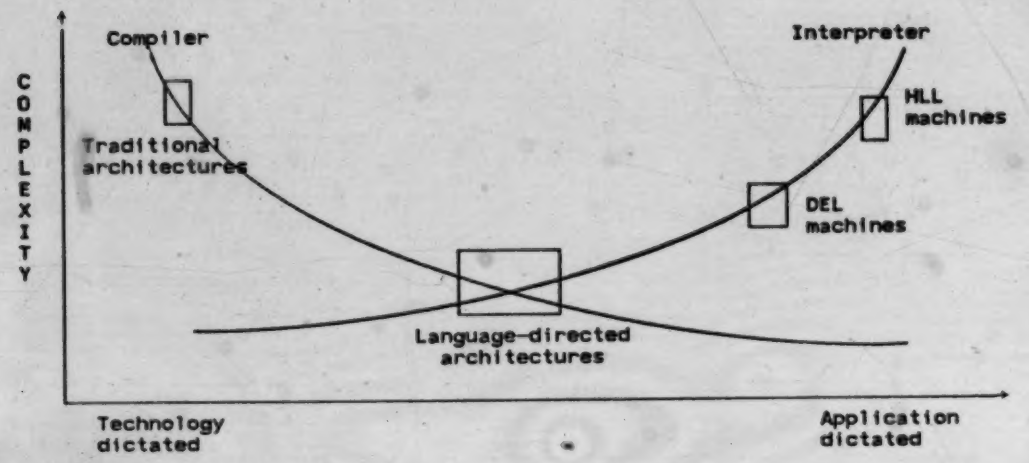

Fig. 1 Complexity vs Design Direction

factors dictated by the high-level language characteristics and the microarchitectural attributes of the host machine.

Figure 1 shows the overall complexity relations between these three possibilities.

This thesis describes an experiment in designing a language-directed architecture realized on a microprogrammable machine. The use of a rigorous methodology and newly developed architectural research tools to design, document and implement the architecture is of central importance. The interconnections of the various components used to realize this architecture through microprogramming are shown as Figure 2.

As mentioned earlier, *both* the high-level language, *L1*, and the host architecture, *H1*, are the principal inputs into the initial design specifications. After the design is formalized (a language is being developed to describe these *formal specifications* [dasg81b]) the entire architecture is described using the architectural description language S*A. The *S*A description* is then translated into a microcoded implementation using the microprogramming schema S* [dasg78,dasg80] for the particular host - S*(H1) [klas81]. The microprograms thus obtained are compiled using the *S*(H1) Compiler* [klas81a], compacted [ride81], and loaded into the hosts control memory realizing the target architecture. (Note that some of the components of Figure 2 are still under consideration or represent related research not described here.)

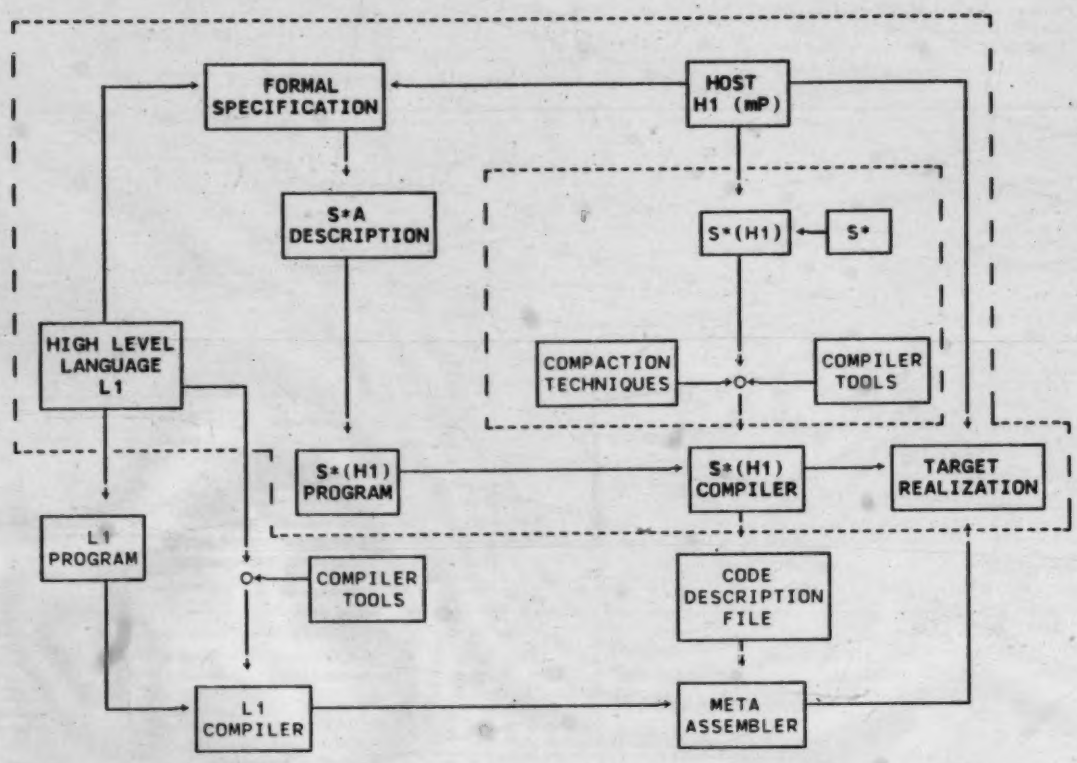

Fig. 2 The Development of a language-directed architecture

This thesis is concerned with a specific instance of the process shown in Figure 2, namely the design of the QM-C – an instruction set architecture oriented toward efficient execution of the programming language C [kern78] (language L1). The QM-C is intended to be realized as a micromachine on the Nanodata QM-1 [nano79] (host H1). Figure 3 shows how the QM-C is related to the QM-1. The QM-C will reside in the QM-1's *nano-store*, and programs written for the QM-C will be compiled into *control-store*. The QM-C treats QM-1's highest level memory, *main-store* as an auxiliary memory.

The design process of the QM-C is similar, in many ways, to that of designing any conventional software system:

1. Based on an extensive study of the usage of the programming language C and the

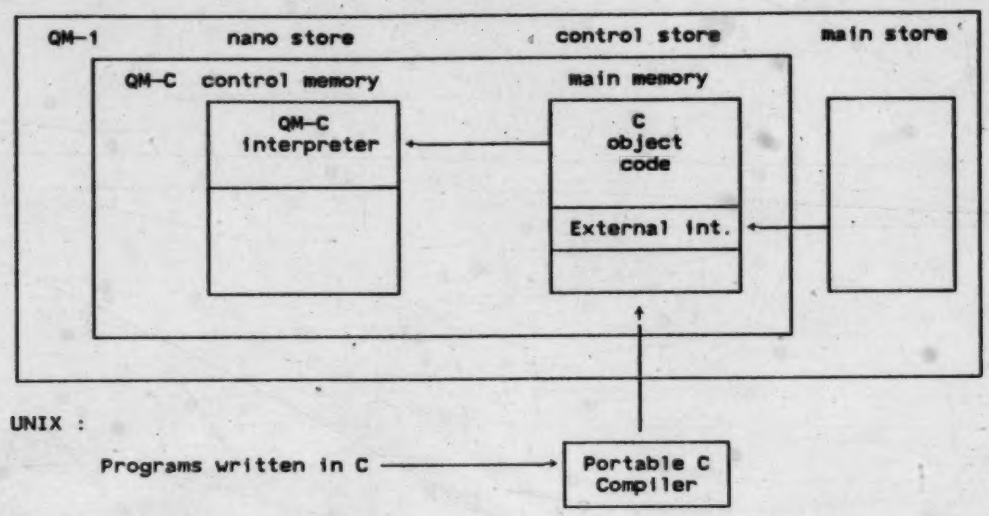

Fig. 3 Usage of the QM-C

QM-1 nano-architecture, design specifications are formalized. This was done by instrumenting the *Portable C Compiler*[2] [john78] to gather data on programs as a part of the compilation process.

2. A high-level modular language (S*A) is used to rigorously describe the design. Its importance here is that it allows the use of top-down design and iterative refinement.
3. The design description is transformed into a specification in a high-level modular implementation language (S*(QM-1)). This transformation was greatly eased because of the similarity or 'kinship' between the description language S*A and the implementation language S*(QM-1).
4. A compiler produces the microcode for realizing the target architecture. (This phase awaits the completion of the compaction and code-generation parts of the S*(QM-1) compiler.)

The remainder of this chapter reviews related research on language directed architectures. Chapter 2 presents the design methodology and alternate design methods are compared. A technique for studying the C language is described in chapter 3 and

---

[2] This compiler is also intended to form the L1 compiler shown in Figure 2.

results of such a study are presented. Chapter 4 describes the QM-C execution architecture. Major design decisions are presented and justified. The impact and usage of the two architectural research tools S*A and S* are briefly discussed in chapter 5. Finally, conclusions and suggestions on future extensions to this work are presented.

1.1 Related Work

Language-directed architectures have been objects of interest for many years. Numerous proposals have been made and many designs completed, at least on paper. This section focuses on designs that take existing programming languages and attempt to define architectures on which to efficiently execute them or their intermediate languages. Mainly, architectures oriented toward the programming language C will be discussed.

Almost all of the more widely used high-level languages have been used as a basis for language-directed architectures, either realized through microprogramming or directly in hardware. The Burroughs corporation is one of few commercial computer manufacturers committed to designing architectures oriented toward the execution of high-level languages in general, specifically Algol. Burroughs also pioneered the concept of dynamically reconfiguring (through microprogramming) their architectures to support many languages [orga78].

An important work in language-directed architectural design is Hoevel's work on a FORTRAN oriented *directly executable language*, DELtran. More important than the actual design is the rigorous method proposed by Hoevel, and the model he developed as the "ideal" directly executable (or intermediate) language for high-level languages [hove78].

Design of PL-1 directed architectures has attracted considerable interest, probably because an architecture oriented toward efficient execution of PL-1, theoretically should perform well for Algol, COBOL and even FORTRAN. Thus, such an architecture would be oriented toward a family of languages. One PL-1 dictated architecture is the *L-machine* [wade72]. This machine is of interest in the context of the QM-C as its direction toward PL-1 consisted of replacing the instruction set of a

conventional architecture with a PL-1 directed intermediate language.

Perhaps the most important of the PL-1 architectures is the *Student PL-1 Machine* designed by Wortman [wort72,myer78]. The importance of his work lies in the method of iterative refinement and top-down approach to the architectural design. Wortman also advocates the importance of empirically studying the performance of the architecture and the necessity for the architecture to be extensible on the basis of these studies.

Other languages that have been used as a basis for language oriented machine designs include COBOL [shap78], Pascal [jone80] and BASIC [burk78]. The programming language C has been attracting increasing interest recently. Presented here are three recent designs all directed in some way towards more effective execution of C programs. Many of the features characteristic of these designs were independently conceived in the QM-C design, a factor that enhances ones confidence in the QM-C approach in general.

The *C-machine* (C/70) built by the BBN company is, at this time, the only commercially available system oriented toward the execution of the C language [kral80]. The C/70 is one of several application-oriented architectures realized through microprogramming on BBN's *Microprogrammable Building Block* (MBB) [kral80]. This computer system was designed to enable a wide range of architectural emulations. The features of the MBB relevant to its usage as a host for language directed architectures are, primarily: a large register file for fast context-switching; the usage of *target-specific* hardware to aid in instruction decode and memory addressing (plug-in boards); and finally, the usage of large *dispatch memory* to facilitate multi-way branching on the basis of instruction parameters.

Unfortunately, not much information is available on the design of the C/70. The instruction set reportedly combines conventional instructions with a rich set of addressing modes to support the reference types of higher level data-structures in C. Thus, the instruction set has approximately 40 instructions with 19 addressing modes plus 44 instructions with restricted addressing (equivalent to at least 800 different

opcodes).

Two documented facts about the C/70 architecture support the results presented in the QM-C design. First, the C/70 architects considered the design of the procedure call/return mechanism to be of central importance [kral80]. The large number of registers in the C/70 enabled the design of a very efficient procedure-call mechanism that simply designates a new set of registers for each process to use, thus virtually eliminating the register save/restore overhead. The second fact is that the instruction set claims to be *"an intermediate language tailored both to the high-level language it is compiled from and the microarchitecture that implements it"*, which is exactly one of the prime design goals for the QM-C instruction set.

S.C. Johnson presented in [john79] a paper design of a C-oriented 32-bit architecture. This machine is the result of over a dozen *iterations* starting from a conventional 16-bit architecture. A compiler was constructed at each stage (using the Portable C Compiler as a base) and the performance measured on a *"large body of C programs"*. The final design turned out to be a very simple instruction set, *"not too heavily slanted toward the C language"*.

The Johnson machine is a two-operand, three-address-mode architecture in its basic form. The addressing modes are "register", "register+offset" and "immediate" for constants. Activation records for procedures reside on a stack in memory and only registers actually used are saved on the stack. C programs for Johnson's 32-bit machine appear typically 10% smaller than for the 16-bit PDP-11 – according to Johnson, attributable largely to the iterative method used in designing the machine.

An alternative to the trend towards increasingly complex instruction sets is offered by Patterson in the design of the RISC I (Reduced Instruction Set Computer) [patt80, patt81]. The purpose of his work is to design a language-directed architecture by moving only the most time-critical features (and those not easily handled by compilers) of the language into firmware (actually RISC-I is intended to be implemented using VLSI technology) leaving the rest up to the software on the system. Although this approach represents almost the opposite extreme to HLLM's discussed earlier, it does

present some interesting points.

According to Patterson, the most time-critical factor in any programming language is the procedure call/return mechanism (this is certainly true for C; see Chapter 3). Thus, RISC's major language-directed component is the architecture of the call/return mechanism. The proposed method is the use of *overlapping register windows*. The saving and restoring of registers is accomplished the same way as in the BBN's C-machine and parameter passing is done by overlapping the register sets of the "caller" with that of the "callee".

Apart form the call/return mechanism the RISC I has a very simple instruction set architecture. All functional instructions work only on registers and memory access is performed using "load" and "store" instructions using one addressing mode (register+offset).

Preliminary performance evaluation expectedly reports increased code size over the architectures to which the RISC I was compared [3]. However, simulated benchmark runs seem to indicate slightly higher performance in speed [patt81].

Although the RISC architecture represents an interesting alternative, care must be taken not to subject the design to be dictated by technological considerations only. An intuitively more attractive alternative is a compromise between the available technology and "ideal" architecture as deduced from the intended application [hove74a]. To attain this compromise, a rigorous design method must be used. The method must discipline the trade-off process between the technological aspects of the host machine and the ideal architecture for the language. The next chapter describes the method employed to provide this discipline in the design of the QM-C.

---

[3]The VAX-11 and the PDP-11 architectures

Chapter 2

Methodology of the Design

Perhaps more important than the actual design of the QM-C is the methodology used to arrive at the final design. With the advent of structured programming and software engineering in general, it became clear that in order to design effective reliable systems a rigorous methodology must be followed [raus76].

The design and implementation of the QM-C takes much from the discipline of structured design and software engineering. Essentially a method of top down design is employed, based on extensive collection of data. The data is then used, along with other factors mentioned below, to direct the design decision process. To document the design and to preserve its internal consistency, a high-level modular description language is used. The architectural description thus obtained is then mapped to a high-level microprogramming language for eventual implementation on the host's micro-architecture. The design of the QM-C is a specific instance of this method. The host is the QM-1's nano-architecture and the goal is to design an architecture oriented towards execution of the programming language C. Fig. 4 illustrates the overall design process in this context.

The novelty lies in the usage of closely related, high level languages to facilitate the mapping from the initial design description of the QM-C's architecture to the actual implementation on the QM-1 [dasg81a]. This transformation process will be discussed in more detail later.

For other approaches to the top-down design of architectures, the reader is referred to [myer78,wort72]. As in Johnson's approach [john79] the design presented here should be considered only a first iteration design. Subsequent design revisions should be made only after the Portable C Compiler has been ported to the QM-C and dynamic statistics collected about the machine's performance. This is a separate research problem, beyond the scope of the thesis.

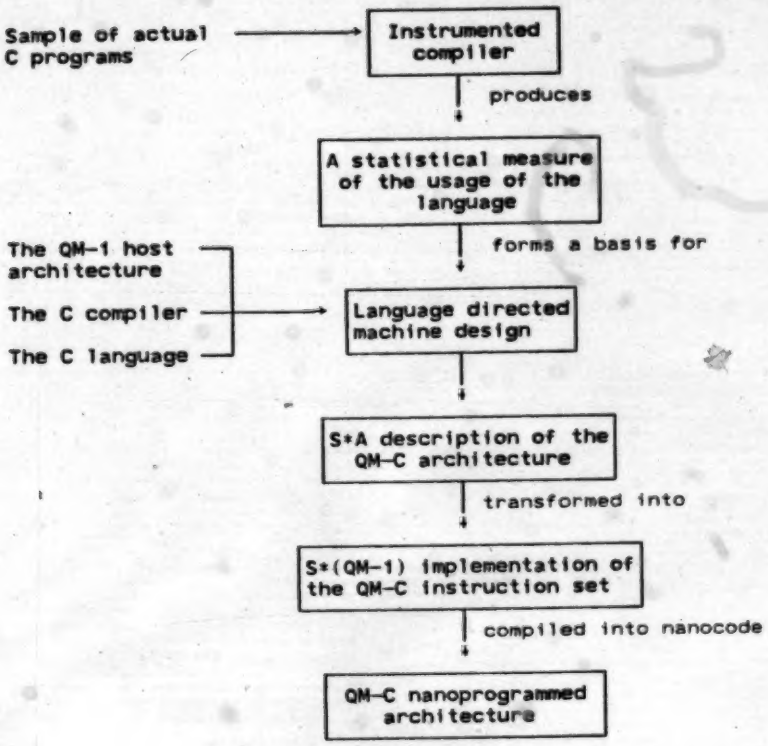
Sample of actual C programs
Instrumented compiler
produces
A statistical measure of the usage of the language
forms a basis for
The QM-1 host architecture
The C compiler
The C language
Language directed machine design
S*A description of the QM-C architecture
transformed into
S*(QM-1) implementation of the QM-C instruction set
compiled into nanocode
QM-C nanoprogrammed architecture

Fig. 4 QM-C's Design Process

2.1 Top-Down vs. Bottom-Up Design

One approach to designing language-directed machine architectures is the *automatic machine tuning technique* [saka79]. Here, statistical analysis is made on the generated code from the target language compiler. Sequences of instructions are counted with the purpose of replacing common instruction sequences with microprogrammed routines. These routines are then added to the machine's basic instruction set. While this *bottom-up* approach may be useful (and may result in improvement) on some machines, it does not apply to user microprogrammable machines like the QM-1. These machines do not have any fixed instruction set, and no resources are dedicated to the support of any one specific machine instruction set. Instead the QM-1 is capable of supporting more general, problem oriented instructions (within the limitations imposed by its basic architecture). The designers of the QA-1, a user-microprogrammable machine similar to the QM-1, reported an experiment where a

conventional instruction set was given as the initial instruction set for the QA-1 and then tuned to a specific application. The resulting tuned architecture achieved less than 10% of the processing power offered by the top-down approach [hagi80].

Thus, for machines like the QM-1, the only possibility is the top-down approach. Based on the statistics on the language, the architect designs the instruction set with the high level language and microarchitecture in mind. The influence of the micro-architecture on the final design might seem to be against the philosophy of a language-directed architectural design. However, ignoring the features and idiosyncrasies of the host will inevitably lead to a reduced performance on that particular host. Only if the host's microarchitecture is very general in nature (bit addressable, possessing powerful mask and rotate mechanisms) can one hope to reduce this influence and still retain acceptable performance. The QM-1 is not such a machine.

2.2 Design Compromise

The design presented here is not intended to be a necessarily optimal *intermediate language* for C. Nor is it intended to be a *canonical intermediate form* [flyn78,hove78] for C. A canonical intermediate form for a high level language is defined to be an instruction set dictated by the language satisfying the following three conditions: (i) There is a one-to-one correspondence between the operations in the language and the operators in the instructions set. (ii) At a particular scope level the number of bits needed to represent each identifier is $\log_2$ of the total number of distinct identifiers. (iii) An instruction executing a high level statement only refers once to each identifier in the statement.

The goal is a compromise between the language's ideal execution architecture, (deduced from its use statistics, available compiler and its inherent structure) and a specific host (the QM-1). The hypothesis is that one can never totally ignore the host's microarchitecture and still implement (through microprogramming) an efficient language directed architecture. The extent of this *host influence* on the overall design depends on the level of the microarchitecture as well as the design goals.

Consequently, a substantial portion of the design of the QM-C is dictated by the QM-1's nano-architecture (especially instruction fetch and decode mechanisms). Other parts of the design (e.g., the method of procedure invocation, and register structure) are dictated by the characteristics of the C language, and still others, (e.g., the functionality of the instructions set and the use of the stack frame) by the observed usage of the language. Finally, though to a lesser extent, the compiler will influence the design.

Thus, the QM-C architecture resulting from the design methodology discussed in this section, can be thought of as a compromise between the various factors which traditionally have been independently used to dictate problem oriented architectural design (e.g., high-level language machines, machine oriented instruction sets, and operating system oriented instruction sets [myer78]).

The next chapter describes an extensive empirical study on the usage of C programs. The results of this study will form the basis on which the most important aspects of the QM-C's architecture will be decided.

Chapter 3

Empirical Study of C Programs

Principal decisions in language-directed machine design should result from extensive use analysis of the programming language. It is usually not possible to directly implement all functional components of the language. Therefore, the designer must decide what to include and how. These decisions must be based on the way the language is actually used as opposed to how it can be used [ditz80a].

This chapter describes a static study of the C programming language done as the first phase in the design of the QM-C's instruction set architecture. On the basis of the results presented here, decisions will be made on what components of the language should be directly supported in the form of instructions, and how the language's abstract execution environment is to be mapped onto the QM-1's nano-architecture.

The most important aspects of the language facing the architect are its functional characteristics (as indicated by the operator and control structure frequencies), its data objects and their interactions, and finally the sizes of various data structures of the language. Thus, measures include operator frequencies, statement types, expression types, typing and storage classes, and stack offsets, to name a few.

In order to make this study as complete as possible, certain aspects of the language were included which might not be of great interest to the machine designer, but would be to compiler writers or software engineers.

3.1 The Sample

Traditionally, statistical analysis of programming languages have based their measures on a relatively small number of what the experimenter considers typical programs [ditz80,elsh76,alex75,robi76]. Since studies of this kind should be as unbiased as possible such predetermination of typical programs can be misleading. *A priori*, the experimenter can not know what are the typical programs written in a particular programming language. Presented here are the results of measurements taken from a

typical *installation*. Thus, one can assume that the study will not be biased toward the experimenter's notion of typical programs nor towards a particular application.

The sample includes all C source programs from a medium sized UNIX environment[4]. It contains the operating system, various compilers, loaders, assemblers, text-formating packages, editors, most utility programs and many user programs. The total sample size is over 2.5M lines of source code in 1705 files containing 8905 procedures.

3.2 Compiler Instrumentation

The *Portable C Compiler* (PCC) written by Johnson [john78, john79a] forms the basis for this experiment. PCC is a two pass compiler roughly 70-80% machine independent. Presented here is only a overview of the compiler; for an excellent, in depth analysis the reader is referred to [leff80]. The first pass of the compiler performs lexical analysis, parsing and symbol table building. It also constructs parse trees for expressions and keeps track of typing. Some code is generated in the first pass (subroutine prologues and epilogues, branch code) but the major code-generation is performed in the second pass. Parsing is done using an automatically generated parser (generated by the UNIX utility YACC [john75]). This parser is a bottom up parser and was easily instrumented to collect statistics on expressions and operators.

Most of the data were collected in the symbol table and tree-building routines. Symbol table management is complicated by the handling of types since in most cases typing is not fixed until all declarations have been handled. Therefore it is necessary to postpone some of the type analysis until tree building was complete. The routines that build the expression trees provide information on expression types, storage classes and variable types. This information is compiled before the complete trees are given to a machine-dependent part for rewriting.

The second (and more machine dependent) pass of the compiler, responsible for code generation, is structured around a goal-directed scheme [aho75]. The compiler

---

[4]A general software research environment on the UNIX operating system [ritc78] at the University of Alberta.

selects one expression at a time from the first pass and consults machine dependent routines to determine the actions required to achieve specified goals. The actual generation of code is performed by matching portions of the expression trees to prestored *templates* [john79a]. When a match occurs, the code portion of the template is written. The tree is then rewritten and the matching repeated until the overall goal is achieved. No statistics were collected at this stage as the characteristics of the code generation routines depend heavily on the target architecture.

The modularity of the compiler and the fact that it is written entirely in C greatly facilitates identifying places where data may be collected. The compiler was instrumented to collect statistics on a per procedure basis and write them to a file after each procedure was compiled. Basically only the machine-independent part of the compiler was instrumented. The machine-dependent portion is modularly separated from the rest and is therefore easily avoided. The only place where the target architecture is directly reflected in the study, is in the stack offset measure (tables 15 and 15a). Even so this measure can still be used as an indication of the stack usage as long as the results are interpreted with the target architecture characteristics in mind.

Operator and statement type frequencies (tables 1,2,3 and 4) were measured directly in the parse routines. Assignment types and expressions (tables 7,8 and 9) were tabulated from the expression trees before going through machine dependent rewriting. Type and storage class analysis was performed during and after tree building (tables 11 and 13). And finally various distribution statistics were computed from the data written by the compiler. The size characteristics of programs are very dependent on the target architecture and therefore largely omitted in this study (except for tables 15 and 15a mentioned above). Counting the number of distinct data objects declared or referenced by a program can, however, give an indication of the storage required for these data objects (e.g. the number of parameters passed to a procedure, or the number of automatic variables declared by a program, gives an indication of the stack behaviour).

3.2.1 Sources of Error

Special care must be taken when interpreting results from any study on programming languages, especially if these results are to form a basis for the architectural decision process [ditz80]. A number of misinterpretations are possible. Firstly, the programming language might not have certain high-level constructs (for example, the lack of I/O statements in C) thus forcing the programmer to implement those using some other mechanism provided in the language (for example, procedure calls). For C then, our statistics indicate the frequency of procedure calls in general, instead of reflecting heavy usage of input or output constructs. As the purpose is not to redesign the language, this potential misinterpretation is ignored.

Secondly, features in the language may be transformed early in the compilation process to a common lower level function. This low level function is then reflected in the measurement taken instead of the higher level feature. The solution to this problem is to do the analysis at a sufficiently high level (i.e before these transformations take place). One might argue that this is not a serious problem. These transformations are easily handled by the compiler (as sometimes in C) without any loss of efficiency, the architect should well concentrate on implementing the more common lower level function. For example:

> References to array elements are translated early in the compilation to a 'base plus offset' reference to a pointer. Thus, a programmer may declare an array; 'int x[10];' and subsequently refer to its i'th element as 'x[i]' in one place and as a pointer reference, '*(x+i)' elsewhere in his program. The compiler immediately translates the conventional array reference 'x[i]' to the equivalent '*(x+i)'.

Finally, one source of error peculiar to this study is the fact that many programs written in C use a preprocessor to include a separate file of external declarations. These declarations may be common to many procedures but not used in their entirety in each one. Although many programs were eliminated from the study because of this, some of the results are slightly biased toward external variables. (tables 11,13 and 14)

3.3 Operators

When orienting an instruction set towards a particular language, great importance must be placed on how the high-level operators are implemented in the target instruction set. If a particular high-level operation (for example "exclusive or") is not used frequently, it does not make much sense to design an instruction to directly implement this operation. Conversely, addition might be heavily used and thus should be fully supported by the instruction set.

Table 1 lists the arithmetic operators in C in order of frequency. Notably, the increment operator is the most common arithmetic operator in C. This is consistent with

Table 1 - Arithmetic Operators

Operation	#	%	Cumul. %
++ (increment)	13596	32.7	32.7
+	8723	21.0	53.6
& (bitwise and)	5931	14.3	67.9
-	5044	12.1	80.0
*	1944	4.7	84.7
-- (decrement)	1780	4.3	89.0
/	1610	3.9	92.8
<< (left shift)	810	1.9	94.8
| (bitwise or)	753	1.8	96.6
>> (right shift)	665	1.6	98.2
% (mod)	577	1.4	99.6
~ (complement)	132	.3	99.9
^ (excl. or)	48	.1	100.0
Total	41613	100.	

studies of languages that lack an explicit increment operator in that addition is usually listed as the most common operation and statements of the form 'a = a+1' as very common assignment types [knut71,alex75]. C is characterized as being rich in operators. This richness seems to be appreciated by programmers and adding explicit increment / decrement operators to programming languages would probably be beneficial. One factor that reflects the usage of C as a system implementation language is the frequency of the bitwise and/or operators. These operators are also used heavily by many other programs, for the purpose of flagging and masking flags.

Relational operators are listed in Table 2; the ordering here is predictable. Notice that relational and logical operators outnumber the arithmetic operators in the C programs studied. This is partly due to the peculiar assignment operators discussed later and partly to the fact that traditionly C has been used infrequently for numerical or statistical computations.

Table 2 - Relational and Logical Operators

Operation	#	%	Cumul. %
== (equal)	17979	35.2	35.2
!= (not equal)	6721	13.2	48.3
&& (logical AND)	6459	12.6	61.0
<	4956	9.7	70.7
|| (logical OR)	3821	7.5	78.1
! (NOT)	3377	6.6	84.7
>=	3326	6.5	91.3
>	2879	5.6	96.9
<=	1592	3.1	100.0
Total :	51110	100.	

Combining the data in tables 1 and 7 gives some indication of expression complexity, but only for arithmetic expressions involved in assignment statements. This gives approximately 0.64 arithmetic operators per expression. Relational and logical expression complexity can be estimated by combining the data in Table 2 with the frequency counts for statements requiring relational expressions (Table 4). This results in 1.29 operators/expression.

These estimates agree reasonably well with previous studies; e.g., Tanenbaum's figure of .45 operators/arithmetic expression and 1.22 operators/relational expression [tane78]; and for XPL [alex75], .41 and 1.19 respectively for arithmetic and relational operators per expression.

As C is the only high-level language incorporating many different assignment operators, it was of interest to measure the usage of these operators (Table 3). In C, in stead of writing **'a = a op b'** one writes **'a op= b'**, where **'op'** is an arithmetic operator. (In fact ++ and -- can be thought of as assignment operators as well.) Assignments of the form 'op=' account for 14.3% of all assignments; this agrees roughly with other studies that have given the frequency of statements of the form **'a = a op expression'** [robi76,tane78,alex75].

3.4 Statements

An important consideration when designing an instruction set is the implementation of the high-level languages control structures in the underlying instruction set. From Table 4 it is evident that the single most important control statement in C is the procedure call. One out of every four executable statements in C is a procedure call, reflecting the high degree of modularity of C programs in general. Table 5 better illustrates the frequent occurrence of procedure calls in C. In 25.5% of all procedures, the ratio of procedure calls to the number of executable statements is less than 1.5, and in only 17% is this ratio greater than 6. This confirms the suspicions of C compiler writers and others that substantial time, in any installation, is spent in the calling mechanism [john79,sutp79]. Procedure calls account for 26.4% of executable statements in C. The average length of a procedure reflects this modularity of programs; each

Table 3 - Assignment Operators

Operation	#	%	Cumul. %
=	57124	86.7	86.7
+=	3719	5.6	92.3
-=	3055	4.6	96.9
|=	973	1.5	98.4
&=	472	.7	99.1
*=	130	.2	99.3
>>=	106	.2	99.5
/=	100	.2	99.7
<<=	90	.1	99.8
^=	85	.1	99.9
%=	67	.1	100.0
Total :	65921	100.	

procedure has on the average 22.7 executable statements. (This compares favorably with 28.6 for XPL [alex75] 18.2 for SAL [tane78] and 86.3 for FORTRAN [knut71])

Another interesting fact about C is that a procedure has on the average 1.27 explicit return statements even though C procedures have an implicit return at the end of the procedure. Of those, 77.3% are used to return a value to the caller.

Of the conditional statements the **'if'** statement is predictably the most common. If statements are generally simple, of the form **'if (condition) stmt;'**. Of the loops **'for'** and **'while'** statements are predictably the most common. The **'do .. while'** loop is almost non existent, indicating that loops evaluating their terminating condition before executing the body of the loop are more useful.

Table 4 - Statement Types

Statement	#	%	Cumul. %
Assignment	81297	40.1	40.1
Proc. call (parametrized)	46435	22.9	63.1
if	25925	12.8	75.9
return(*exp.*)	8761	4.3	80.2
Proc. call (no parameters)	7186	3.5	83.7
break	5757	2.8	86.6
else	5331	2.6	89.2
for	4520	2.2	91.4
while	4224	2.1	93.5
goto	3186	1.6	95.1
?: (shorthand if)	3091	1.5	96.6
return	2575	1.3	97.9
continue	2388	1.2	99.1
switch	1359	.7	99.7
do .. while	548	.3	100.0
Total :	202583	100.	

Table 5 - Average Number of Statements Between Procedure Calls

# statements	# procedures	%	Cumul. %
(1.0,1.5]	2270	25.5	25.5
(1.5,2.0]	1297	14.6	40.1
(2.0,2.5]	685	7.7	47.8
(2.5,3.0]	938	10.5	58.3
(3.0,3.5]	383	4.3	62.6
(3.5,4.0]	651	7.3	69.9
(4.0,4.5]	189	2.1	72.0
(4.5,5.0]	474	5.3	77.3
(5.0,5.5]	127	1.4	78.7
(5.5,6.0]	380	4.3	83.0
(6.0, -]	1511	17.0	100.0
Total :	8905	100.	

The static frequency of the '**switch**' statement is surprisingly low. Table 6 shows the '**switch**' statement in more detail as its dynamic frequency is expected to be quite high. The primary interest when designing a possible implementation of the '**switch**' statement is the average number of cases. From Table 6 we see that 99.8% of all switch statements have fewer than 64 cases, and that 69.7% have fewer than 8 cases. Thus, if the switch statement is to be implemented using a table-driven branch, for example, the instruction should preferably allow different table sizes, ranging from 2 to 64 addresses.

As in other programming languages that have been analysed [ditz80, tane78, alex75] the assignment statement accounts for approximately 40% of all executable

Table 6 - Switch Statements

# of cases	statements	%	Cumul. %
2	31	2.3	2.3
(2,4]	398	29.2	31.5
(4,8]	519	38.2	69.7
(8,16]	293	21.6	91.2
(16,32]	98	7.2	98.5
(32,64]	18	1.3	99.8
(64,-]	3	.2	100.0
Total :	1359	100.	

statements[5]. It was therefore decided to analyse assignment statements separately and the results of that study are shown as Table 7. These results are specially relevant to decisions concerning instruction formats and addressing modes. In the context of Tables 7, 8 and 9, the following points should be noted :

1. *stack* represents variables declared as automatics or as parameters to a procedure. That is, variables referenced off of a stack.
2. *address* represents variables declared as external or static. That is, variables referenced by their static address.
3. *register* represents local variables declared to reside in fast storage.

For example the entry *stack address* in Table 7, indicates the frequency of direct assignment to a local variable (automatic or parameter) from an external (or static) variable.

Some interesting conclusions may be drawn from Table 7. (a) Simple assignments of the form 'a op= b' account for 59.3% of all assignments whereas assignments of the form 'a op= expression' are 40.7%. (b) Constant assignment account for 35.5% of all

[5] Assignment statements in Table 4 include the increment and decrement operators (++ and --).

assignments (c) 53% of all assignments are to local variables.

Of interest in this context are the kind of expressions, both arithmetic and logical commonly formed by C programmers. Table 8 shows the frequency of the various types of arithmetic expressions. These statistics exhibit many of the same characteristics as the assignment statements. Expressions of the form **'variable op constant'** account for 77.5% of all expressions. Expressions involving local variables are the most common single group, 21.1% of all expressions.

Note that one expression may contribute many times to this table. The table is compiled directly from expression trees (such that an expression like '**a+b-2**' would contribute once to '**variable op expr.**', depending on how '**a**' was declared and once to '**expr. op constant**').

These results are an important consideration when designing the addressing modes and the operands for selected instructions. Table 9 lists the same data for relational and logical expressions.

Data, similar to those presented in tables 7, 8 and 9 is, to the authors knowledge, not available on other languages. That, along with the fact that these tables are very language-dependent inhibits detailed expression- or statement-type comparison between C and other languages.

As is apparent from tables 7, 8 and 9 constants play an important role in C (note that character constants are not distinguished from other constants - the compiler immediately transforms character constants to their numerical equivalent). Table 10 gives the distribution for constant magnitudes. Similar to what has been observed in other studies [alex75] 52.8% of all constants are zero, one or two.

Table 7 - Assignment Types

lvalue = rvalue	#	% in group	% of total
stack constant	13554	38.2	20.6
stack address	1059	3.0	1.6
stack register	1261	3.6	1.9
stack stack	6080	17.2	9.2
stack *expr.*	13457	38.0	20.4
Total :	35411	100.	53.7
address constant	4967	39.8	7.5
address address	765	6.1	1.2
address register	751	6.0	1.1
address stack	1247	10.0	1.9
address *expr.*	4747	38.0	7.2
Total :	12477	100.	18.9
register constant	4856	26.9	7.4
register address	803	4.5	1.2
register register	644	3.6	1.0
register stack	3097	17.2	4.7
register *expr.*	8633	47.9	13.1
Total :	18033	100.	27.3
Gr. Total :	65921		100.

Table 8 - Arithmetic Expression Types

expr. arop expr.	#	% in group	% of total
stack constant	14226	79.0	16.6
stack address	176	1.0	.2
stack register	82	.5	.1
stack stack	1058	5.9	1.2
stack *expr.*	2472	13.7	2.9
Total :	18014	100.	21.1
address constant	2796	52.2	3.3
address address	326	6.1	.4
address register	29	.5	0.0
address stack	90	1.7	.1
address *expr.*	2119	39.5	2.5
Total :	5360	100.	6.3
register constant	9707	90.7	11.3
register address	47	.4	.1
register register	169	1.6	.2
register stack	109	1.0	.1
register *expr.*	673	6.3	.8
Total :	10705	100.	12.5
expr. constant	39609	77.0	46.3
expr. address	328	.6	.4
expr. register	263	.5	.3
expr. stack	797	1.5	.9
expr. expr.	10474	20.3	12.2
Total :	51471	100.	60.2
Gr. Total :	85550		100.

Table 9 - Relational and Logical Expression Types

expr. op expr.	#	% in group	% of total
stack constant	13873	74.4	27.9
stack address	563	3.0	1.1
stack register	192	1.0	.4
stack stack	2350	12.6	4.7
stack *expr.*	1681	9.0	3.4
Total :	18659	100.	37.5
address constant	2870	60.3	5.8
address address	659	13.8	1.3
address register	44	.9	.1
address stack	126	2.6	.3
address *expr.*	1061	22.3	2.1
Total :	4760	100.	9.6
register constant	4568	59.5	9.2
register address	629	8.2	1.3
register register	553	7.2	1.1
register stack	602	7.8	1.2
register *expr.*	1327	17.3	2.7
Total :	7679	100.	15.4
expr. constant	7457	40.0	15.0
expr. address	437	2.3	.9
expr. register	262	1.4	.5
expr. stack	650	3.5	1.3
expr. *expr.*	9840	52.8	19.8
Total :	18646	100.	37.5
Gr. Total :	49709		100.

Table 10 - Distribution of Constant Magnitudes

Constant	#	%	Cumul. %
0	19303	17.1	17.1
1	23958	21.2	38.3
2	16409	14.5	52.8
(2,4]	7679	6.8	59.6
(4,8]	8438	7.5	67.1
(8,16]	8694	7.7	74.8
(16,32]	4363	3.9	78.7
(32,64]	4221	3.7	82.4
(64,128]	3283	2.9	85.3
(128,256]	1275	1.1	86.4
(256,512]	10119	9.0	95.4
(512,1024]	1427	1.3	96.7
(1024,2048]	240	.2	96.9
(2048,4096]	515	.5	97.3
(4096,-]	3010	2.7	100.0
Total :	112934	100.	

3.5 Data Types and Storage Classes

Support for high-level types and the mapping of the language's storage classes to the target architecture is an important consideration for the efficiency of the resulting design. C has thirteen basic types for variables, listed along with their observed frequencies in Table 11. This rich typing capability appears somewhat underutilized in the actual usage of the language. *Integers, characters* and *structures* account for 84.8% of all objects declared in C. This does not mean that the other types in the language are dispensable. The implication is that the architect of a language-directed instruction set

Table 11 - Basic Types

Type	#	%	Cumul. %
Integer	66528	46.6	46.6
Character	31844	22.3	68.9
Structure	22684	15.9	84.8
Long	7580	5.3	90.1
Short	6958	4.9	95.0
Unsigned int.	2983	2.1	97.1
Union	2395	1.7	98.8
Double	1151	.8	99.6
Unsigned short	403	.3	99.9
Float	119	.1	100.0
Unsigned char	32	0.0	100.0
Unsigned long	14	0.0	100.0
Enumeration	1	0.0	100.0
Total :	142692	100.	

Table 12 - Typing Complexity

Typing Complexity	#procedures	%	Cumul. %
(0,0.5]	8155	91.6	91.6
(0.5,1.0]	527	5.9	97.5
(1.0,1.5]	104	1.2	98.7
(1.5,2.0]	68	.8	99.4
(2.0, -]	51	.6	100.0
Total :	8905	100.	

Table 13 - Storage Classes

Class	#	%	Cumul. %
External	70658	49.5	49.5
Structure member	29123	20.4	60.9
Register	10792	7.6	77.5
Automatic	10268	7.2	84.7
Parameter	9222	6.5	91.2
Structure name	5123	3.6	94.8
Type def.	3801	2.7	97.5
Union member	1498	1.0	99.5
Label	1377	1.0	99.5
Static	721	.4	99.9
Union name	104	.1	100.0
Others	5	0.0	100.0
Total :	142692	100.	

might decide to map some of these less used types into those that are more commonly used.

Table 12 shows the *typing complexity*. In C the number of types is potentially infinite as basic types can be augmented with modifiers to produce *pointers* to a type, *functions* returning an object of some type, and *arrays* of objects of some type. The typing complexity is calculated as :

$$\frac{\text{The number of array, pointer and function modifiers}}{\text{The number of basic types declared}}$$

and thus reflects the frequency of such type constructs. For example, if a procedure declares two variables; **int** x[10] and **char** *y the resulting typing complexity for the

procedure would be 1. If the declaration for y were changed to **char** y, the typing complexity would reduce to 0.5. As expected the overwhelming majority of procedures (91.6%) have typing complexity in the range 0. - .5.

Storage class usage is tabulated in Table 13. As aforementioned, the storage class usage is an important input to the design of the addressing modes and the general execution environment. Although externals are by far the most common storage class, the exact number is probably slightly overestimated because of the "include facility" in the C programming environment. These numbers also reflect the importance of structures in C. Structure declarations outnumber array declarations almost 2 to 1, indicating that structures are a more important high-level data object than arrays (contrary to statements in [myer78]). Any instruction set oriented toward the C programming language, must provide a way to handle structures efficiently.

Another important measure of the usage of storage classes is their *reference density* (i.e. the average number of references to each variable declared). Table 14 displays the reference density of the elementary storage classes (global, local and local register). Also of interest is the reference density of the higher order data objects. These are shown as Table 14a. Note that structures are included here with the higher order data object, even though structures are considered basic types. Note also that the numbers for arrays and pointers in Table 14a are based on directly observed frequency of pointer declarations (40342) and array declarations (14315), but, as mentioned earlier, the same data object might be used as a pointer in one place and an array elsewhere in the same program. Thus these numbers are not exact and serve only as indicators on the respective usage of these types.

3.6 Size Characteristics

Next to implementing the functional characteristics of a high-level language in the instruction set architecture, the most important consideration is the organization of the space occupied by a program in its execution environment. In C this involves the design of the stack frame and decisions on how to best implement the static data space.

Table 14 - Reference Density

Reference to	#	%	Density
Register var.	76051	25.6	7.0
Automatics and parameters (i.e. off the stack)	80693	27.2	4.1
Externals and statics (i.e. by address)	139833	47.1	2.0
Total :	296577	100.	

Table 14a - Reference to Complex Types

Reference to	#	%	Density
Pointers	75421	39.6	1.9
Arrays	60592	31.9	4.2
Structures	34975	18.4	1.5
Strings	19236	10.1	1.0
Total :	190224	100.	

Tables 15 and 15a show the distribution of the stack offset size for the procedures in the sample. The C compiler pushes arguments to procedures onto the stack before saving the calling procedure's environment. Next, space is allocated for the called procedure's local variables (automatics). These numbers are in terms of eight bit bytes and computed from code produced for an architecture in which characters occupy 8 bits, integers occupy 16 bits and long integers 32 bits, and are therefore influenced by the target architecture. Despite that, general conclusions can be drawn about the typical size of the stack frame.

A more natural measure of the usage of the stack frame can be obtained by counting the number of objects residing on the stack. Table 16 shows the distribution of the number of parameters to called procedures. These numbers are averaged over all procedure calls within each procedure. This shows that the average call is to a procedure with roughly 1.5 parameters. The difference in distribution from Table 16 to Table 16a (number of parameters per procedure defined) is probably caused by two things. Firstly, many procedure calls in C are to system routines not included in the sample (notably I/O routines) and secondly, parametrized procedures are usually called several times in a program (with different parameters), while procedures with no arguments are used primarily to modularize the program, and thus can be expected to have a lower static frequency.

Table 17 tabulates the same data for local (automatic) variables. The simplicity of procedures in C now becomes quite apparent. Only 2.1% of all procedures in the sample have more than 8 automatic variables. As mentioned earlier, C provides the programmer a means to specify the fact that frequently used local variables should reside in registers. Table 18 shows that approximately 54% of the procedures declare one or more register variables. Combining the data on local variables form tables 17 and 18, the result is similar to Tanenbaum's 3% of procedures declaring more than 11 local variables.

The fact that C programmers are aware that they are programming for a machine with three variable registers, is reflected by the data of Table 18. Thus, it is not possible to make any significant conclusion about the optimal number of variable registers needed in C. It has been argued [myer78, tane78] that because of the overhead in saving and

Table 15 - Stack Offset (arguments)

Offset in bytes	# procedures	%	Cumul. %
0	2557	28.7	28.7
2	3040	34.1	62.9
4	2160	24.3	87.1
(4,8]	934	10.5	97.6
(8,16]	211	2.4	100.0
(16,32]	3	0.0	100.0
(32, -]	0	0.0	100.0
Total :	8905	100.	

Table 15a - Stack Offset (automatics+temp.)

Offset in bytes	# procedures	%	Cumul. %
0	5085	57.1	57.1
2	848	9.5	66.6
4	767	8.6	75.2
(4,8]	678	7.6	82.9
(8,16]	561	6.3	89.2
(16,32]	276	3.1	92.3
(32,64]	148	1.7	93.9
(64,128]	106	1.2	95.1
(128,-]	436	4.9	100.0
Total :	8905	100.	

Table 16 - Average Number of Parameters per Procedure called

Average #	# procedures	%	Cumul. %
0	400	4.5	4.5
(0,1]	2245	25.2	29.7
(1,2]	3461	38.9	68.6
(2,3]	1783	20.0	88.6
(3,4]	583	6.5	95.1
(4,5]	203	2.3	97.4
(5,6]	109	1.2	98.6
(6,-]	121	1.4	100.0
Total :	8905	100.	

Table 16a - Number of Parameters per Procedure Defined

# of parameters	# procedures	%	Cumul. %
0	2557	28.7	28.7
1	3154	35.4	64.1
2	2139	24.0	88.2
3	694	7.8	95.9
4	227	2.5	98.5
5	75	.8	99.3
6	45	.5	99.8
>6	14	.2	100.0
Total :	8905	100.	

Table 17 - Distribution of Automatic Variables Defined

# automatics	# procedures	%	Cumul. %
0	5561	62.4	62.4
1	1369	15.4	77.8
2	638	7.2	85.0
3	379	4.3	89.2
4	281	3.2	92.4
5	186	2.1	94.5
6	135	1.5	96.0
7	96	1.1	97.1
8	74	.8	97.9
>8	186	2.1	100
Total :	8905	100.	

Table 18 - Distribution of Register Variables Defined

# Reg. Variables	# procedures	%	Cumul. %
0	4087	45.9	45.9
1	1458	16.4	62.3
2	1367	15.4	77.6
3	1608	18.1	95.7
4	240	2.7	98.4
5	71	.8	99.2
6	59	.7	99.9
7	13	.1	100.0
8	2	0.0	100.0
Total :	8905	100.	

restoring registers, local variables should never be placed in registers; or even that registers should not be used at all. While this may be true of other languages it is certainly not true of C. The programmer usually has a pretty good idea which variables are the most heavily referenced (observe the register variable reference density in Table 14) and thus should reside in a limited faster storage. C provides the programmer with the ability to convey this information to the compiler. The execution architecture should have a way of utilizing this information so that programs may be executed more efficiently. The burden of making sure that this potential is not lost due to overhead in saving and restoring used registers falls on the architect of the procedure calling mechanism.

3.7 Summary

We have provided, in this section, a comprehensive study of the characteristics of the C programming language, how the language is used, and how C programmers make use of the facilities provided in the language.

The overall conclusion one may draw from this study is that C programs are essentially simple. Procedures are generally short, composed of simple statements organized in a simple structured manner. The inherent complexity of the problems (or algorithms) coded in C appears to be distributed through the interaction between relatively simple modules. C programmers build complex programs from small modules with well defined interfaces. Thus, the basic philosophy of the UNIX system is strongly carried into the way the C language is used.

The next chapter describes how the data discussed here were used as the basis for the design of the QM-C's instruction set and execution environment.

Chapter 4

The QM-C Execution Architecture

In this chapter the overall design of the QM-C is described. The factors determining the final design are described and the justifications provided for crucial design decisions. As explained in Chapter 2, the design presented here is a compromise between the "ideal" execution architecture for C (as deduced from the use-statistics on the language) and the natural emulator environment of the QM-1.

The architecture of the QM-1 was designed to emulate traditional machine architectures using a unique three-level interpretation. The approach combines the programming ease of vertical *microcode* with the efficiency of horizontal *nanocode*. The emulator resides in the QM-1's *control-store* coded in a vertical, machine-oriented microinstruction set (MULTI [burh78]). The microinstructions are in turn interpreted by the horizontal nanocode in *nano-store*. Programs for the emulated machine reside in the highest level memory – *main-store* [nano79]. Figure 5 shows the traditional emulator environment on the QM-1.

The purpose here is to replace the nano-routines that implement the vertical micro-instruction set with a nano-coded interpreter that interprets a language oriented instruction set. This allows programmers to write programs (emulators) for the QM-1 in C and subsequently compile them into control-store. Figure 6 shows how the QM-1 would function as a dedicated satellite processor enabling architectural experiments to be carried out in the UNIX/QM-1 environment. Substantial portions of emulators and *Emulator Control Programs* [demc76] could now be written in C instead of the low-level, machine oriented MULTI.

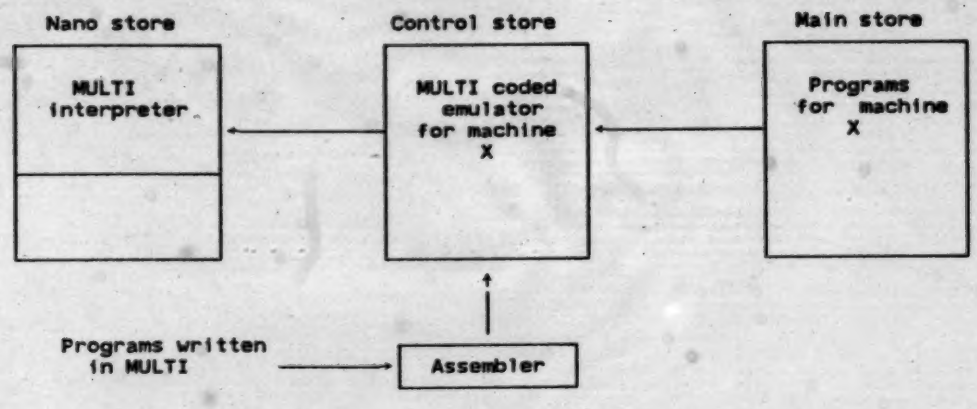

Fig. 5 QM-1's Emulator Environment

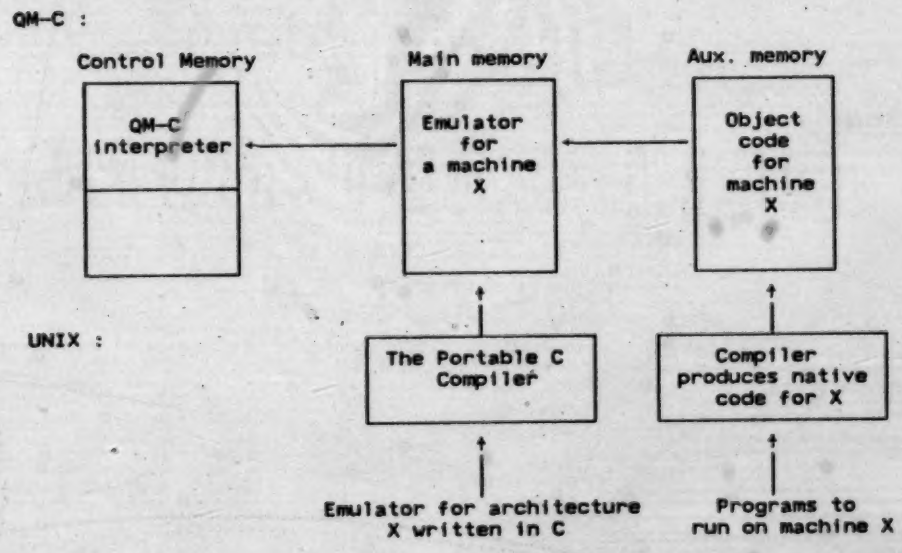

Fig. 6 QM-C/UNIX Environment

4.1 Design Alternatives

Designing and implementing microprogrammed architectures presents some unique problems. The architect's "freedom of choice" is necessarily limited by the characteristics of the host architecture (e.g. word-length, memory organization, alu- and shifting capabilities etc.). The QM-1's nano-architecture was specifically designed to

interpret micro-instructions in which general emulators would be written. As a consequence, many assumptions are made about the format of control store instructions and hardware resources provided to efficiently interpret an instruction set conforming to these assumptions. It is of course possible to bypass these fixed resources and implement their counterparts in nanocode, resulting in a larger, more complicated interpreter.

Thus we are faced with two design alternatives :

I. using the QM-1's nano-architectural resources where ever possible, consequently limiting our freedom in designing instruction formats, addressing modes and data representations to what is "acceptable" to the QM-1. The advantage is greater speed and smaller simpler interpreter at the cost of larger code representation.

II. implementing the instruction set processor entirely in nanocode. This would allow greater freedom in designing instruction formats (instruction fetch/decode in nanocode) and place fewer restrictions on the data representation. The advantage here is possibly smaller program representation, at the cost of a slower, more complicated, interpreter.

The decision on these alternatives heavily influences the subsequent design of the QM-C architecture. Both possibilities must therefore be given careful consideration.

Let us first consider the second alternative. Here the QM-C's main memory (QM-1's control store) is viewed as a single stream of bits; instructions and data representations are of arbitrary (possibly variable) formats. Instruction execution involves substantial decoding to identify the operation and locate the operands. The execution loop can be divided into three steps :

(a) Instruction Fetch. Control store must be read as 18-bit words, and the beginning and end of instructions located.

(b) Instruction Decode. Locate operands in the instruction, generate a nanostore address and transfer control to the routine implementing the instruction.

(c) Instruction Execution. Calculate effective address of operands, and actually execute the instruction.

For the QM-1, step (a) would require reading 18-bit words out of control store, into a

buffer of some sort, checking each one for the end of instruction. This would involve using the QM-1's primary alu and shifter mechanisms. Step (b) would also require the shifter and the alu to locate the operands, and generate the nano store address. Thus, both steps (a) and (b) use the same resources and can not be performed in parallel. A conservative estimate is that the implementation of steps (a) and (b) would occupy at least two words of nanostore on the QM-1. As most instructions use the alu or the shifter (or both), performing step (a) in parallel with execution would be impossible.

For alternative I, QM-C's main memory is viewed as an array of 18-bit words. The hard wired micro-instruction execution mechanism is used to decode instructions. This places restrictions on the instruction formats (7-bit opcodes, restricted combination of 5,6,11,12 or 18-bit operands, 18-bit instructions followed by arbitrary number of 18-bit extension words) and basically only one data-type, the 18-bit word.

As before, the instruction execution mechanism can be divided into three stages. Step (a), the instruction fetch, now consists only of reading a word from control store into a dedicated instruction buffer. No QM-1 resources are used, apart from the control store bus. Instruction decode is handled entirely in hardware and consists of constructing the nanostore address on the basis of the first 7 bits of the word on the control store bus (the opcode), and placing the rest, the operands, into the *instruction register*. This whole process is initiated by simple primitives and uses neither the shifter nor the alu. Thus, during the execution of an instruction i (say) the decoding of instruction (i+1) can be initiated as well as the fetch of instruction (i+2).

As the primary design goal for the QM-C was the speed of execution (possibly at the cost of code-size) it was decided to make use of the nano-architectural conventions of the QM-1 (alternative I). This represents the largest factor in the "host influence" on the QM-C architecture. Even if this decision limits the extent to which the QM-C can be directed towards the C language, there is still much room for the language's influence to be reflected in the QM-C architecture.

In subsequent sections a C oriented instruction set will be designed, but *within the limitations imposed by the QM-1's nano-architecture.*

4.2 Storage Organization

A good storage organization (or *name-space*) is characterized by a simple correspondence between the source name-space (in order to simplify compilation and preserve transparency) and the hosts name-space [hove78].

In C there are three inherently distinct classes of storage : *External* (or global) variables (i.e., variables, whose lifetime extends throughout the program's execution), *automatic* (or local) variables that are declared and allocated storage in each procedure, and the third class, peculiar to C, *registers*. The register storage holds variables declared by the programmer to be heavily used and therefore should reside in a fast (limited) storage. Register variables are always local.

Mapping these storage classes to the QM-1 name-space does not present serious difficulties. Global variables will reside in QM-C's main memory (QM-1's control-store) and will be referenced by their static 18-bit address. Register variables map naturally into the QM-1's register file (local-store). Local-store is a file of 32 general-purpose 18-bit registers. Of those eight will be used to hold variables, four will be used for expression evaluation and other temporary storage. The rest are dedicated to specific purposes (described later) or are not used by the QM-C. Figure 7 shows how the QM-C's register file maps into local-store on the QM-1. Automatic variables must be mapped into dynamic storage. The general consensus is that the best way to implement the dynamic storage of block structured languages is via some kind of stack mechanism [john79,keed78].

The QM-1 has no stack-management primitives but they can easily be implemented in nano-code. It was therefore decided to implement a stack in QM-C's main memory to hold local variables and other dynamic objects. As is evident from the use-statistics of C, one of the most important aspects of the QM-C design is the organization of the dynamic storage area for procedures (the *stack-frame*). The stack will be used to hold the automatic variables, and parameters to procedures; the contents of the registers must also be saved on the stack when context switching occurs.

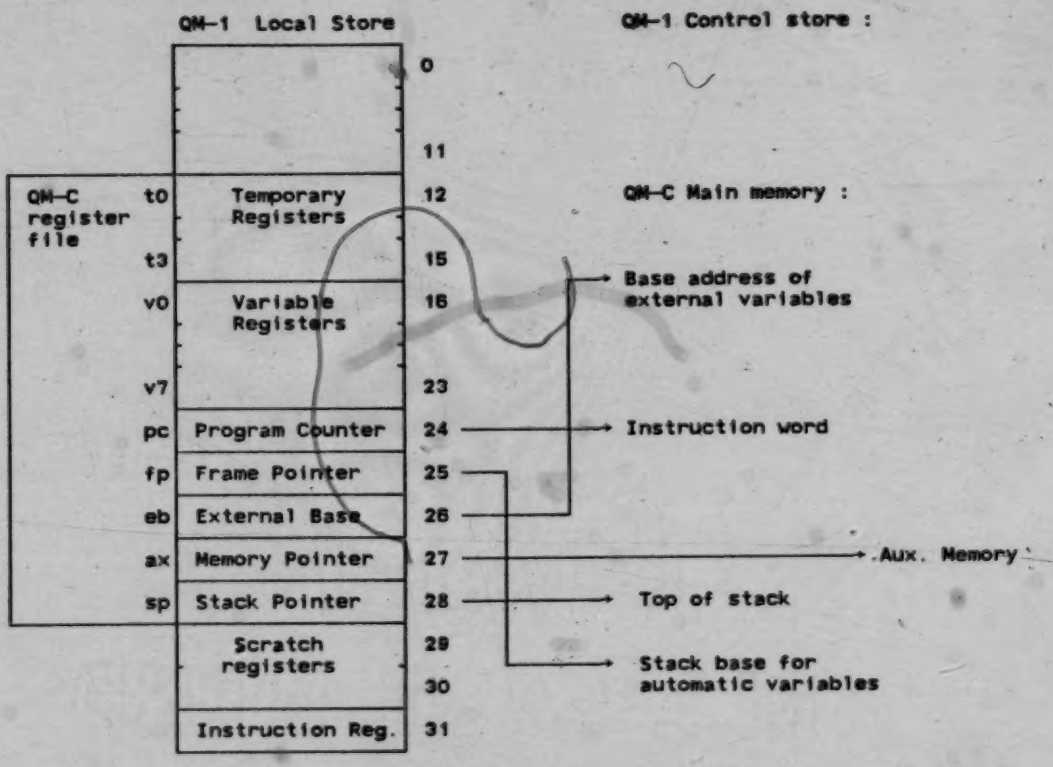

Fig. 7 QM-C's Register File

The design of the QM-C's stack-frame is similar to that of the VAX 11/780 [stre78] but somewhat simpler. The calling procedure begins the call by pushing arguments to the called procedure onto the stack. When all arguments have been pushed the "call" instruction is executed. After calculating the effective address of the called procedure its *mask-word* is read. This is the first word of all procedures and contains the lowest register number used by the called procedure along with the size of the stack area needed for local variables. On the basis of this information registers are saved and storage for automatics is allocated on the stack. Finally control is transferred to the first instruction of the called procedure. Figure 8 shows a snapshot of the top of the stack, just after a procedure A has called a procedure B with two parameters.

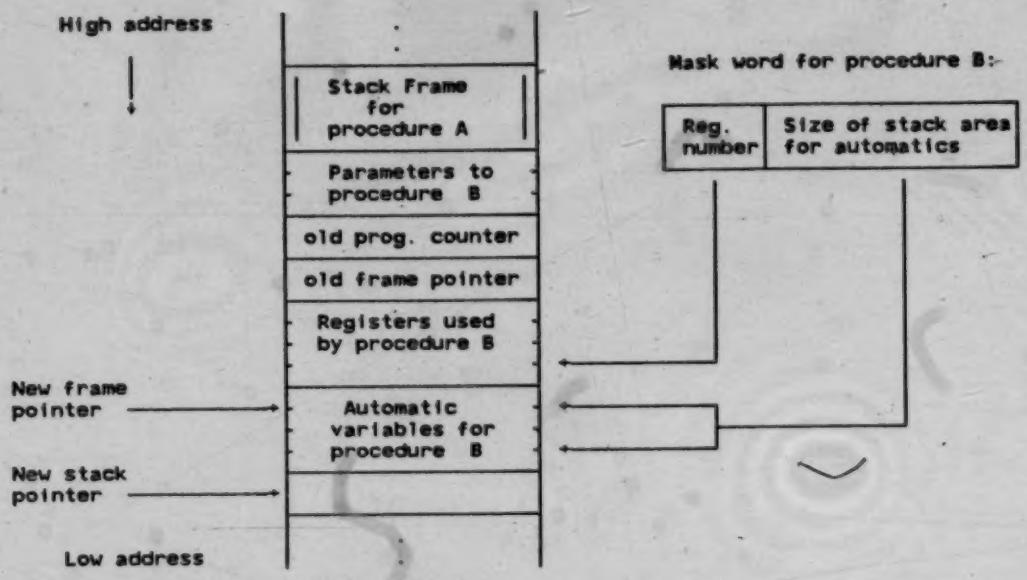

Fig. 8 QM-C's Stack Mechanism

When the called procedure returns, essentially the reverse of the aforementioned events takes place. The return instruction has two operands; the register number as in the mask-word and the size of the argument list. The registers are "popped" from the stack, including the old frame-pointer. The last thing done before control is returned to the caller is the deallocation of the argument area from the stack. (This can be done at this point since all arguments in C are passed by value.)

Looking at the use-statistics for C, it can be seen that approximately 95% of all procedures can now refer to their parameters *and* their local variables using a 6-bit signed offset from the frame-pointer. This design of the stack-frame also makes it possible to increase the maximum number of registers available to the programmer as programs that declare few register variables are not penalized by the overhead of saving unused registers.

4.3 Addressing Modes

So far we have decided to partition the QM-C name-space into three distinct classes of storage; static storage (for external global variables), dynamic storage (for automatic or local variables) implemented on the stack, and register storage. Now the task is to decide how to reference the values of the variables residing in these different storage classes.

One of the consequences of the decision to use the QM-1's fixed instruction decode mechanism is the limited number of distinct opcodes available (128). Of primary importance, therefore, is to minimize the number of different addressing modes. The reason for this is the requirement that the QM-C instruction set be as close to *complete* as possible, to minimize the generation of non-functional move instructions. An instruction set is "complete" if all its instructions are "complete". An instruction is said to be "complete" if it is capable of operating on any combination of the different addressing modes defined for its operands. (For example, a "complete" two operand add instruction in an instruction set with three classes of storage (A, B and C), each referenced via different addressing, would require nine formats; add A to A, add A to B, add A to C, add B to A, etc.. If, however, classes B and C (say) were addressed using the same mechanism, the number of required formats would reduce to four.)

The design of the stack-frame dictates that automatic variables and parameters be referenced as offsets from the address contained in the frame-pointer (i.e. a "register+offset" type of reference). Therefore it was decided to implement the same method of reference for external variables. (This is similar to Johnson's approach [john79].) Thus one register is reserved to hold the base address of external variables (the *external-base* register in Figure 7).

Because of the high frequency of relatively small constants in C's expressions and statements, it was decided, in order to make the best utilization of instruction formats, to encode constants directly in the instruction.

The resulting design now has three different reference methods : (i) **register+offset** for external, automatic and parameter variables, (ii) **register** for

automatic register variables and (iii) **immediate** for constant operands. At this point a decision must be made on how the various typing information will affect the organization and reference to the three basic storage classes.

4.4 Type Mapping

One of the restricting aspects of the QM-1 architecture is its addressability. Both control store and main store are organized as arrays of 18-bit words addressable with 18-bit addresses. This places serious limitations on the mapping from the basic types in C to the QM-C architecture. Following is a discussion and justification of the overall characteristics of this mapping.

As the QM-1 has no mechanism to handle floating point data efficiently and since the types *float* and *double* are relatively unimportant in C, it was decided not to support these types at all in the QM-C. The relative infrequency of unsigned types and the limited opcode-space of the QM-C resulted in the decision not to separately support these types – but to map them (through the compiler) into their signed counterparts. The basic types *structure* and *union* are essentially composite types; unions are handled entirely by the compiler. Structures will be discussed later.

It remains to show how *integers, characters, longs* and *shorts* are mapped into the uniform 18-bit word on the QM-1. It was decided to map these four types all into the 18-bit basic storage element of the QM-1. The rationale for this decision is given below :

(a) **Opcodes.** By making no distinction, in the QM-C instruction set, between integers and characters results in a significant saving in the opcodes needed to support these types. (For example, separate move-character and move-integer instructions are no longer needed.)

(b) **Use-statistics.** Long integers represent only 5.3% of all variables in our sample, where integers occupy 16-bit words. It is therefore reasonable to assume even lower percentage for an architecture with 18-bit basic integers. The opcode argument also applies here since no separate instructions are needed to handle long integers. (Note that, if needed, long integers could still be implemented by the

compiler writer using procedure calls.)

(c) **Homogeneity.** As a result of the uniformity of the name-space, all conversion operators are eliminated. This is especially important in C because of the relatively heavy use of pointer variables. If characters and "short" integers where implemented as 9-bit quantities, two per word (as would be natural to preserve the transparency of the mapping) and "longs" as 36-bit integers, pointers to these basic types would also have different representations. (Character pointers would have to include indication of which byte in a word contains the character, values of long pointers would have to be divisible by two, etc.)

Also, if characters occupy 9-bits, that would have to be the smallest addressable entity of the QM-C. The result would be numerous allingment restrictions and would probably necessitate the introduction of 9-bit registers, thus destroying the homogeneity of the register-file.

(d) **Transparency.** It is a potentially serious design flaw to provide direct support (in the form of instructions) for data types (such as short integers) when the host machine would execute such instructions slower than their "full-integer" equivalents. When a programmer declares variables as short integers to conserve space, and subsequently observes a reduction in the overall efficiency of the code, there arises a strong incentive to use only ordinary integers. This is true even if the problem's data structures map more naturally into short integers.[6] A better solution is to equip the compiler to map "shorts" into ordinary integers such that the programmer is not aware of this mapping but does not experience any reduction in efficiency attributable to the usage of short integers. Programs are thus closer in representation to the actual problem, and when moved to a machine that supports these data-types, make full use of available resources. Thus, the conclusion is that if a microprogrammable host architecture is heavily slanted toward a particular basic storage element, the best choice is to make this the smallest addressable entity of the target architecture.[7]

---

[6] This is the approach taken in the PDP-11 C compiler.
[7] Hoevel concludes similarly in his work on DEL's [hove78,hove74a]

The primary disadvantage of this many-to-one type mapping is, of course, the increased size of the program's data-space. This increase is specially significant for *character strings* in the QM-C (they will occupy twice the space actually needed). This is, however, somewhat compensated by the elimination of type and pointer conversion operators from the code-space.

The overall assessment of the merit of this decision on the quality of the QM-C execution architecture must await a dynamic study of the code generated for the machine and its consideration in subsequent design iterations.

4.4.1 Structures, Strings, Arrays and Pointers

The C compiler and the defined characteristics of the language dictate largely the implementation of these composite types. In C, array references and pointer references are indistinguishable (as explained earlier). A reference to an array element (for example **a[i]**) is transformed immediately by the compiler to a pointer reference of the form ***(a+i)**. This direct correspondence is used by programmers at the source level in C.

Since both external and automatic variables are referred to via the "register+offset" addressing, the decision is to refer to arrays using the same method. And as mentioned earlier, the array concept in C maps well into this mode of addressing. As strings in C are considered to be arrays of characters and are treated as such at the source level, no additional support is provided for strings.

Structures are the most important composite type in C (a structure is considered a basic type by the compiler, because variables can be of type structure, and can be augmented with type modifiers to produce arrays of structures and pointers to structures). Pointers to structures dictated the decision to support structures with the same mechanism as arrays (again "register+offset"). It was found that 77.8% of all structure references were indirect references (i.e. members referred to via an offset from a pointer variable containing the address of the structure). This fact along with the limited opcode-space available for the QM-C, dictates the "register+offset" support for structures as well. Also, examination of the Portable C Compiler reviled that a substantial change in the machine independent portion would be needed, if structures are to be

handled in a significantly different manner [john79a].

4.5 Instruction Formats

As a consequence of the decision to use the hard-wired instruction decode mechanism of the QM-1, the bit layouts of instructions are totally dictated by the host machine.

The nano-architecture provides the capability to directly invoke nano-routines on the basis of instruction words in control-store. This mechanism uses the first 7 bits of a control-store word to construct a nano-store address. The remaining 11 bits are transferred into a special register (the instruction register) where they can be accessed as either one 11 bit field or a 5 bit field followed by a 6 bit field. The nano-routine implementing the instruction may read subsequent words out of control-store, possibly into the instruction register for decoding. A simplified schema of the instruction decode mechanism is shown as Figure 9.

This mechanism imposes the following limitations on the format of the instructions :

(a) 7 bit opcodes. Maximum number of opcodes is 128.

(b) Operand fields can be 5,6,11,12 or 18 bits long.

(c) One word instructions restricted to

  i 2 operand fields : 5 and 6 bits long.

  ii 1 operand fields : 11 bits long.

(d) Two (or more) word instructions have, in addition, the possibility of interpreting the second (and subsequent) word as containing

  i 3 operand fields : all 6 bits long.

  ii 2 operand fields : 6 bits and 12 bits.

  iii 1 operand field : 18 bits long.

The difficulty now is to map the three previously defined addressing modes of the QM-C into these restricted operand formats. Before describing this mapping, a discussion about the number of operands per instruction, is in order.

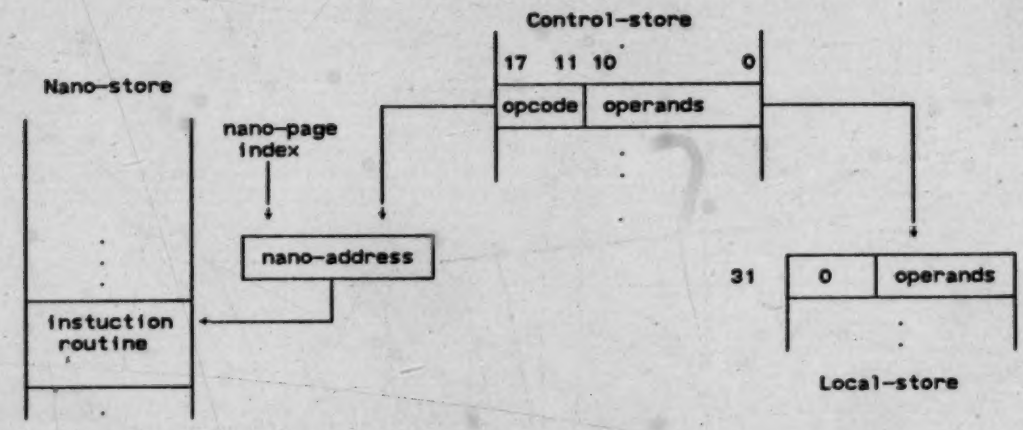

Fig. 9 Instruction Decode Mechanism

At first glance it might be concluded that in order to minimize the generation of nonfunctional move instructions, a three operand instruction set would be ideal (i.e. perform the operation on two operands and store the result in the third). There are basically two situations where three operand instructions would save space:

1. A three operand form of

   a = b op c

   would in most cases be smaller than the move, op, move sequence on a two operand machine.

2. Evaluating complex expressions would require less code since two operands could be directly computed into temporary storage.

However, the decision for the QM-C was to design instructions taking one or two operands. There are three major reasons for this.

Firstly, studies conducted by other researchers indicate that savings in number of instructions generated for a three operand machine are lost because of larger instructions [shus78, john79]. Johnson, for example examined the effect of turning his 32 bit architecture into a three operand machine. He observed a reduction in code size of approximately 4% [john79]. The static measurements on C and other languages seem to provide the reason. The average complexity of expressions is very low. The

measurements on C did not include separate measures for arithmetic expression complexity, but similar languages usually exhibit .41 to .45 arithmetic operators per expression [alex75,tane78].

Secondly, mapping a three operand instruction into the restricted instruction formats mentioned previously would be more difficult. One word instructions would be eliminated. For example, a three operand instruction working on registers only, would take two words and waste 12 bits.

Finally the most decisive argument against the three operand QM-C is the limited opcode space. For the QM-C, a "complete" (in the sense explained before) three operand instruction set requires up to triple the amount of opcodes as the same two operand instruction set.

It is now possible to identify six basic formats for "complete" two operand instructions, using the three previously defined methods of reference :

1. Constant to Memory variable (global or local) (**CM**). Operation performed on a constant and memory variable and the result put back into memory.
2. Constant to Register (**CR**).
3. Register to Register (**RR**).
4. Register to Memory (**RM**).
5. Memory to Register (**MR**).
6. Memory to Memory (**MM**).

In Table 19 the possible mappings of these basic formats into the operand formats dictated by the QM-1 instruction decode mechanism, are tabulated.

General references to memory are, as aforementioned, all of the form "register+offset". Since the most frequent references are directly to external or automatic variables, a considerable saving in terms of size (and sometimes time) can be achieved by designing separate instructions to handle implicit stack offsets and external offsets respectively. The possibility of optimizing constant and offset field sizes also exists. These are listed in Table 19 under "Possible Opt.".

Basic Format	Operand Fields	Words	Possible Opt.	Operand fields	Words
Constant to memory (CM)	c11, Rb, o12	2	Constant to extern. (CE)	c5, o6	1
	c6, Rb, o18	2		c11,o18	2
	c18, Rb, o6	2	Constant to stack (CS)	c5, o6	1
				c11,o18	2
Constant to register (CR)	c18, R	2	-	-	-
	c6, R	1			
Register to register (RR)	R, R	1	-	-	-
Register to memory (RM)	R, Rb, o18	2	Register to extern. (RE)	R, o6	1
			Register to stack (RS)	R, o6	1
Memory to register (MR)	Rb, o18, R	2	Extern. to register (ER)	o6, R	1
			Stack to register (SR)	o6, R	1
Memory to memory (MM)	Rb,o6,Rb,o12	2	Extern. to Memory (EM)	o6,Rb,o18	2
				o11,Rb,o12	2
			Stack to Memory (SM)	o6,Rb,o18	2
				o11,Rb,o12	2

Legend :
R - Register cN - N-bit constant
Rb - Base register oN - N-bit offset

Table 19 - Instruction Formats

4.6 Operator Selection

The method of selecting operators is derived from Hoevel's work on Directly Executable Languages [hove78,hove74a].

Assume that we have a set $F = \{f(1), f(2), \ldots f(n)\}$ of potential operators. The task is to decide which of the operators f(i) should be candidates for direct implementation as nano-coded instructions x(i) and which should be implemented entirely using other operators $\{f(j),\ j \neq i\}$, applying the idea of *macro-expansion* X(i) of f(i). We wish to determine a subset G of F that in some sense minimizes a significant percentage of the code. We chose to select the members of G on the basis of our study on the C code base. The task now becomes simply to select the most common operators in C in order of their frequency until either we run out of nano-store or have exhausted the set F. In light of the limited opcode-space it is clear that we will run out of opcode-space long before the set F is exhausted. The remaining operators in F must then be implemented

with macro-expansion using the already selected operators.

4.6.1 Reduced vs Complex Instruction Set

The limited number of opcodes available to the QM-C instruction set complicates the selection of operators for the set G. At first glance it might be concluded that the frequency of some of the higher-level constructs in C (for example the **for** loop) dictates reservation of opcodes to support their execution. – The extent of the language support might even be measured in terms of such direct implementation of the higher level language constructs. This is not true for the QM-C.

Rather than reserving an opcode for the direct implementation of statements like **for** loops, it was decided that macro-expansion would take care of their implementation and use the opcode thus freed to provide size or speed (or both) optimizations to the more frequent (simple) instructions. (For example, adding a constant to a register is performed at least 6 times more often than **for** loops; thus using the freed opcode to provide a one word optimization of this instruction would result in more savings than an instruction supporting the execution of **for** loops.)

One of the effects of this method of selecting operators, is that the resulting instruction set becomes simpler and closer to what in known as a *Reduced Instruction Set* (RIS) [patt80,patt81]. Another argument for selecting a RIS for the QM-C is the pipelined nature of the instruction execution mechanism; the time taken to execute n one-word instructions (in a straight line segment) can be essentially the same as executing one n-word instruction. Thus if the macro-expansion of one n-word instruction is n one-word instructions, nothing is gained but one precious opcode is "wasted". (This can happen on the QM-C because of the restricted format of instructions.)

In the next section an overview of the instructions is given – their function, operands and number of opcodes provided for each.

4.7 Instruction Set Overview

The QM-C instruction set may be partitioned into three classes similar to Flynn [flyn78]; functional (F-type), procedural (P-type) and move (M-type) instructions. The class of functional instructions includes instructions that actually perform operations on the programs variables, such as addition, subtraction etc.. The P-type instructions are the branch, procedure call/return and related instructions. The M-type move data in memory and between memory hierarchies. Note that the M-type instructions are not necessarily "overhead" instructions [flyn78]. Substantial number of M-type instructions are typically generated directly from a program's source statements (simple assignments for example). Similarly, F-type instructions can sometimes be classified as pure overhead (array index calculations, character manipulations, etc.).

This partitioning of the instruction set is done mainly for convenience and is not intended to be used directly for measuring architectural efficiency (as in Flynn [flyn78]). The ratios between the number of instructions generated from each class can only be used to obtain performance information if taken together with source-code measurements that identify overhead instructions from any of the three classes.

4.7.1 Functional Instructions (F-type)

Increment/Decrement: These operations collectively account for 37% of all arithmetic operations in C. They are inherently one-operand instructions, but for some format combinations a second operand, - containing the increment - is provided. The 'INC' and 'DEC' instructions have, in addition to their two basic formats (R and M), five optimized formats :

G1 - general memory variable, short offset, incremented/decremented by 1

E1, S1 - external or stack variable, long offset, incremented/decremented by 1

EC, SC - external or stack variable, short offset, incremented/decremented by a 6-bit unsigned constant

Addition / Subtraction: These operations account for 33.1% of C's arithmetic operators

and therefore have full, 6-form addressing generality (CR, CM, RM, MR, RR, and MM). In addition to the basic forms the 'ADD' and 'SUB' instructions have six optimized forms :

ER, SR - external or stack variable to register

RE, RS - register to external or stack variable

EM, SM - external or stack variable to general memory variable

And / Or: These operations account for 16.1% of the arithmetic operators and receive the same addressing generality as addition and subtraction. Since the constant operand to these instructions must be 18 bits, two different optimized formats were added to make a total of eight :

CE, CS - 18-bit constant to external or stack variable

Left and Right arithmetic shift: As these instructions account for only 3.6% of the arithmetic operators they receive restricted basic addressing (RR, CR, RM and CM) and no optimizations.

Multiplication / Division : Although these operations are more common in C than the shifting operations (10% of total arithmetic operations), the inherent difficulty in performing these operations on the QM-1 lead to a register only support. (The time taken to move variables into registers is negligible compared to the time it takes to perform the operations.) Hence, the 'MUL' and 'DIV' instructions exist only in the RR format.

The rest of C's arithmetic and logical operators are handled by a generalized *ALU-instruction* with restricted addressing (RR).

4.7.2 Procedural Instructions (P-type)

There are basically two alternatives that should be considered when designing conditional branch instructions :

1. Maintain a set of *condition codes* and design a separate *compare* instruction to set these codes on the basis of its two operands.
2. Embed the comparison in the branch instructions making them into three operand instructions. This would eliminate the need for a separate compare instruction and hide the condition codes completely.

The limited number of opcodes available on the QM-C together with the fact that the

QM-1 supports direct condition code setting as a side-effect of all alu and shifter operations, dictates the choice in these matters. Rather than providing multiple format branch instructions (two or more words each) a four-format *compare* instruction was designed. All the conditional branch instructions now become one-format, one-operand instructions saving a number of opcodes.

Since all F-type instructions implicitly set the condition codes, one might conclude that a separate compare instruction would not be needed. However, subtracting two values (without 'gating' the result into a register) is a considerably faster operation on the QM-1 than subtraction into a temporary location. Care must be taken to design the condition code management in a regular coherent manner [russ78]. It was decided to set the condition codes only as a result of the execution of F-type instructions. Thus, no M or P-type instructions affect the condition codes and the setting is consistent for all F-type operations (i.e. adding one to a variable results in the same setting as incrementing the same variable etc.)

Branch Instructions : Four conditional branch instructions are provided. They are designed on the basis of the four most common relational operators; 'JMPEQ' (==); 'JMPNE' (!=); 'JMPGT' (>) and 'JMPLE' (<=), which collectively account for 88% of all the relational operators. These branches are relative to the program counter and enable distances of up to 1024 words. One unconditional relative branch instruction is provided (JMPR) as well as two forms of an absolute branch instruction; 'JMPA' for short distance branches and 'JMPAL' for long branches.

Because of the suspected high dynamic frequency of the C switch statement, a table-driven multi-way branch instruction is provided. This instruction is capable of absolute branches on the basis of a table (up to 64 words) following it in memory.

Procedure Call Instructions : The call and return instructions, implementing the context-switching mechanism discussed earlier, are probably the most important in the QM-C instruction set. Procedure call and return statements account for 32% of all executable statements in C, more than all other statements (excluding assignment) taken together.

1. **Primary instructions**

```
INC ( M,  R)                          -Increment
DEC ( M,  R)                          -Decrement
PUSH(C,M,P,R)                         -Stack push

ADD(CM,CR,MM,MR,RM,RR      )          -Addition
SUB(CM,CR,MM,MR,RM,RR      )          -Subtraction
AND(CM,CR,MM,MR,RM,RR      )          -Logical and
IOR(CM,CR,MM,MR,RM,RR      )          -Logical inclusive or
RSH(CM,CR,      RM,RR      )          -Right shift
LSH(CM,CR,      RM,RR      )          -Left shift
MUL(              RR       )          -Multiplication
DIV(              RR       )          -Division
ALU(              RR       )          -General alu function

CMP(CM,CR,MM,MR,   RR      )          -Compare
MOV(CM,CR,MM,MR,RM,RR,PR,PM)          -Move

JMP(EQ,GE,LT,NE)                      -Conditional branch
JMP(A, AL, R)                         -Unconditional branch
SWITCH                                -Multi-way branch
CALL                                  -Procedure call
CALLA                                 -Procedure call (long)
RETN                                  -Return
```

2. **Secondary instructions (optimized for speed and size).**

```
INC (E1,EC, G1, S1,SC      )
DEC (E1,EC, G1, S1,SC      )
PUSH(E,          S,   PE,PS)

ADD(      EM,ER,RE,RS,SM,SR)
SUB(      EM,ER,RE,RS,SM,SR)
AND(CE,CS,EM,ER,RE,RS,SM,SR)
IOR(CE,CS,EM,ER,RE,RS,SM,SR)

CMP(      ER,             SR)
MOV(CE,CS,ER,   RE,RS,    SR,SCR)
```

Table 20 - Instruction Set Summary

The 'CALL' instruction takes two forms; long and short absolute address.[1]

The 'RETURN' instruction, as discussed earlier, takes two operands and exists only in one form.

4.7.3 Move Instructions (M-type)

The QM-C has two M-type instructions 'MOV' and 'PUSH'. Movement within the registers, main-memory and between registers and main-memory is accomplished with the 'MOV' instruction. It possesses full 6-form addressing generality, and in addition two forms to move *pointers* to register or to memory (PR, PM). Seven optimizations are

---

[1]The CALL instruction is described formally in Appendix III as an example of the application of the S*A description language. Its S*(QM-1) code is shown in Appendix IV.

provided :

SCR – move short constant to register

ER, SR – move external or stack variable to register

RE, RS – move register to external or stack variable

CE, CS – move constant to external or stack variable

The 'PUSH' instruction is used to place parameters on the stack prior to procedure calls. 'PUSH' is a one–operand instruction with full addressing (R, M and C) and two optimizations :

E, S – push external or stack variable

Pointers may also be directly pushed onto the stack using three other forms :

P – push general pointer variable

PE, PS – push pointer to external or stack variable

The justification for a separate push instruction for procedural parameters comes from the high frequency of parametrized procedure calls; 87% of all calls.

An overview of the QM–C instruction set is shown as Table 20. For a more detailed description of each instruction refer to Appendix I.

In the next chapter we will discuss the architectural description language S*A which was used to formally describe the design presented above, and as a tool in the design process. The microprogramming language S*(QM–1), used to implement the QM–C instruction set, will also be discussed and the interaction between these two languages explored.

Chapter 5 .

On Architectural Description and Implementation

The importance of the two architectural research tools employed in the design and implementation of the QM-C was discussed in chapter two. An integral part of the work presented here was testing the usefulness and validity of these tools.

In any design methodology, the language used to represent and communicate the design, plays a vital role. For the QM-C, the whole architecture was described using the architectural description language S*A [dasg81]. This language is a member of a *family of languages* for the design and implementation of machine architectures. This family also contains the basis for the implementation language - the S* microprogramming language schema. The usage and importance of this *family of languages* has been discussed in detail elsewhere[9], but, in summary, they are :

(a) By expressing the architecture in a formal language, the design may be verified with respect to an initial set of specifications.

(b) The two members of the family used in the QM-C design are used at two distinct levels in the design process. Since the languages are designed to be relatively close to one another, the transformation from description to implementation is relatively painless and mentally manageable. Furthermore, all stages of the design may be documented within a unified framework, and the high level modularity of the S*A description carries over to the actual implementation language.

(c) Rather than define a single, huge description language capable of representing all levels of the design abstraction, the language family contains two compact, closely related languages, one for each of the principal levels of the design process. One thus obtains the distribution of complexity necessary for a structured, top-down design.

In this chapter we will elaborate on the idea of *stepwise refinement*, starting from the S*A description of the QM-C architecture and ending with the realization of this

---

[9] In *"Towards a Family of Languages for the Design and Implementation of Machine Architectures"*, [dasg81a], by Dasgupta and Olafsson. Some portions of this chapter are based on that paper.

architecture, through microprogramming, on the QM-1 host. This process will be illustrated using a small example from the design of the QM-C procedure call instruction.

First a short overview of the relevant characteristics of the S*A description language and the S*(QM-1) microprogramming language will be given[10].

5.1 The S*A Architectural Description Language

The principal constructs in S*A are the *system* and the *mechanism*. These constructs provide the architect means of *modularizing* the design and to study the various components of the architecture in isolation [dasg81].

Thus, an architecture described in S*A is viewed as a collection of systems which may, in turn, be composed of simpler systems. At the lowest level in this hierarchy the systems is composed of one or more basic modules called *mechanisms*.

Basically, an S*A mechanism consists of a set of *global* and *private* state variable declarations, public procedures (that can be invoked from other mechanisms), and private procedures. A procedure in S*A is a description of the data flow sequence that is necessary for realizing a particular firmware function. In Table 2.1 an overview of the characteristics of the S*A language is given. (An example of an S*A description of a hardware component of the QM-C, the SELECT_NEXT procedure describing the QM-1/QM-C instruction decode function, is shown in Appendix III, page 98. Also shown is an example of the usage of S*A to describe an instruction that is nano-coded on the QM-1, the INCM procedure (page 101)).

A mechanism is *active* whenever one of its procedures is being "executed". Thus, a mechanism is activated by calls on its public procedures from procedures inside other mechanisms. Private procedures can only be invoked from procedures inside their own mechanism.

---

[10]For a complete definition of these languages the reader is referred to [dasg81,olaf81] for S*A and [dasg78,klas81a] for S* in general, and [klas81,klas81b] for the instantiated language S*(QM-1)

Characteristics	S*A	S*	S*(QM-1)
Primitive data types	bit	bit	bit
Structured data types	seq, array, tuple stack, assoc array	seq, array, tuple stack, assoc array	seq, array, tuple
Synchronizing Primitives	yes	yes	no
Constant Declaratives	yes	yes	yes
Pseudo Variables	no	yes	yes
Channels	yes	no	no
Basic executional statement	assignment statement	assignment statement schema	machine specific assignment statement
Control Statements	if..fi, while..do, repeat..until, case, call, act, return, exit, goto	if..fi, while..do, repeat..until, case, call, return, goto	if..fi, while..do, repeat..until, case, call, act, return, goto
Parallel Statements	yes	yes	yes
Machine-timing related constructs	None	cocycle..coend, stcycle..stend, region..endreg	cocycle..coend, region..endreg
Modularization concepts	system, mechanism, procedure	program, procedure	program, procedure, macro

Table 21 Language Overview

Mechanisms satisfy the following *mutual exclusion rule*: a public procedure cannot be called when the mechanism containing it is already active. For example, the MEMORY system (whose skeleton is shown in Appendix III, page 103) contains two mechanisms; MAIN__MEM and AUX__MEM. These mechanisms function as *critical regions* [brin77] containing the QM-C memories declared as *private* variables. Thus, accessing the memories is only accomplished by calling one of the global procedures within these mechanisms, and only one access is allowed at one time.

The overall composition of the QM-C system is shown in Appendix III. The system is partioned into subsystems; INSTRUCTION__CYCLE, MEMORY, INTERRUPT and NANO__ARCHITECTURE. The INSTRUCTION__CYCLE subsystem consists of two mechanisms INSTRUCTION__FETCH and INSTRUCTION__DECODE (whose processes are shown in Appendix III) and a subsystem named INSTRUCTION__EXEC. The remaining

principal subsystems of the QM-C are composed of sets of mechanisms, whose details are omitted in Appendix III[11].

The QM-1 resources used by the QM-C are declared as global variables in the S*A description. Some of these are shown in Appendix III as a part of the INSTRUCTION_CYCLE subsystem. The complete set of declarations is essentially the same as the fixed declaration section of S*(QM-1) programs and are omitted here as they are discussed in detail in [klas81a]. The QM-C data objects are declared as *synonyms* of these fixed QM-1 resources.

Many of the data types available in S*A appear in Appendix III. Only some will be discussed here, and for a complete description the reader is referred to [olaf81]. The only primitive data type in S*A is the **bit**, consisting of the values $\{0,1\}$. Bits may be structured into ordered *sequences* or into higher order structures such as *arrays* or *tuples*. Elements of type **array** are accessed by indexing the array name with an integer constant or with the name of some other data object. The **tuple** denotes an ordered collection of components which may themselves be structured. Thus, the variable "*reg*" in the S*A description of the QM-C (a synonym of the global variable denoting QM-1's local store) may refer to its i'th element as "*reg[i]*" or to the j'th *index* register as "*reg.index[j]*".

An important characteristic of the declarations in S*A is the facility to specify alternate data structures for variables. Thus, the variable "*reg*" is declared both as an array and a tuple; the tuple field "*reg.inst_reg*", for example, is further defined in terms of alternate structures.

As mentioned before, the S*A description language is closely related to the S* microprogramming schema, and therefore to its instantiations (see Table 2.1). In the case of the QM-C the description (as represented in S*A) and the implementation in S*(QM-1) share almost identically declared data objects. The difference lies in the operations defined on these objects.

---

[11]The complete S*A description of the QM-C is available from the author.

The next section presents a brief overview of the relevant characteristics of the S*(QM-1). For a complete description the reader is referred to [klas81b].

5.2 The S*(QM-1) Microprogramming Language

The S*(QM-1) is a high level language based on the microprogramming language schema S* [dasg78]. The idea behind S* is that the only way to design a relatively high level microprogramming language is to *partially specify the syntax and semantics of a basic schema* and then obtain the full language by filling in the specifications of the schema on the basis of the idiosyncratic properties of a particular machine. Thus, S* is said to be *instantiated* into a particular language (in this case the S*(QM-1) [klas81,klas81a]).

The fully defined entities of S* include a set of primitive and structured data types, constructs for declaring data objects, and a set of composite statements (the main characteristics of S* are listed in Table 21). Elementary statements are only partially defined since their form and meaning will vary from one microarchitecture to another. Basically then, the instantiation of S* into S*(QM-1) consists of declaring all the QM-1 nano-architectural resources (this declaration forms a fixed, invariant portion of every program) and defining the operations allowed on these data objects, within the S* syntax.

The modularization constructs in S*(QM-1) are *programs*, *procedures* and *macros*. Programs are composed of the aforementioned fixed declaration part, synonym declarations, and a set of procedures of type **instruction**, **subroutine**, and **interrupt**. A program represents a complete abstraction of the contents of the QM-1 nano-store. (See Appendix III for an overview of the program realizing the QM-C architecture). Each procedure is composed of statements whose exact form and functionality is dictated by the QM-1 nanoarchitecture. The basic characteristics of the S*(QM-1) language are summarized in the rightmost column of Table 21.

5.3 Transformation from Description to Implementation

Once the S*A description of the QM-C was complete this *architectural description* had to be transformed or translated to a *microprogram* (nano-program) realizing the architecture on the QM-1. Because of the kinship between S*A and S*(QM-1) this transformation process was greatly facilitated. There are, however, some important factors that must be considered:

(a) The high level modularization constructs in S*A must be transformed to the lower level organization allowed for in S*(QM-1) microprograms. Thus, the **sys/mech/proc** hierarchy of S*A description was reorganized as a set of **proc**'s and **macro**'s within a S*(QM-1) **program**.

(b) The data objects specified in the S*A description are global or local to particular mechanisms. The S*(QM-1) data objects are global to all procedures and are, as aforementioned, a fixed part of any S*(QM-1) program.

Since the same data types exist in both languages the task of mapping the data objects from S*A to S*(QM-1) was greatly facilitated. Also, the QM-C data objects were designed for implementation on the QM-1, making their S*A description almost identical to those of the S*(QM-1).

(c) The S*A description is a specification of the control of information flow and transformation within the QM-C architecture. The procedure in the corresponding S*(QM-1) is a nano-program, i.e., a symbolic representation of the contents of QM-1's nano-store. Thus, the principal difference between a S*A *description* and a S*(QM-1) *implementation* lies in the operations defined on the data objects contributing to the QM-C architecture, as opposed to the data objects themselves.

An example of an instance where substantial refinement of the S*A description is needed to obtain the S*(QM-1) code is in the QM-C multiplication instruction. Since S*A has a richer set of operators, the description of the "MULRR" instruction is essentially of the form :

reg[ ] := reg[ ] * reg[ ]

The S*(QM-1) procedure implementing this instruction is quite involved as the QM-1 has no multiplication primitives, and therefore "MULRR" is implemented using repeated addition and shifting.

The other extreme also exists in the translation process. The S*A description of the condition code mechanism of the QM-C directly describes the underlying QM-1 condition codes. Thus, the S*A procedure that describes the setting of the condition codes as a result of alu- and shifter operations (the SET__STATUS procedure called in the INCM instruction shown in Appendix III), transforms into two assignments to pseudo-variables in S*(QM-1). Thus

alu_status := 1

shift_status := 1

instructs the QM-1 to "gate" the condition codes into a dedicated status register, where they may be tested in subsequent instructions.

To illustrate further the transformation process from an S*A description to a final S*(QM-1) implementation, and to give some idea on how this process might be partially automated, we will in the remainder of this chapter discuss how a portion of the QM-C CALL instruction was transformed from an S*A description (shown in Appendix III) to the final S*(QM-1) microprogram (see Appendix IV). These examples were hand-compiled from the S*(QM-1) code to the intermediate language acceptable to the compaction phase [ride81,ride81a], as the code generation portion of the compiler is still under development [klas81a].

Figure 10a shows the S*A description of the **repeat** statement from the QM-C CALL, responsible for saving the content of the QM-C registers on the stack. The statement loops until the register specified in **inst_reg.c** has been saved. The code includes a call to one of the global procedures in the MAIN__MEM mechanism, to write the register to memory and decrement the stack pointer (sp). To clarify the description, the call to MAIN__MEM.PUSH__G has been expanded in-line in Figure 10b.

In Figure 11a the **repeat** statement has been transformed to the S*(QM-1) representation. The call to the PUSH__G procedure has been replaced by a **macro** call and the syntax of the statement adjusted to the S*(QM-1) syntax. The procedure is then compiled and the semantic analyser would indicate the inability of the compiler to generate code for the **until** condition of the **repeat** statement. The programmer's knowledge of the QM-1's nano-architecture enables him or her to detect the reason; the

```
repeat                                                          10a
     mm_data_select := mm_data_select - 1;
     call MAIN_MEM.PUSH_G;
until  mm_data_select = reg.inst_reg.c;
```

```
repeat                                                          10b
     mm_data_select := mm_data_select - 1;

     do   main_memory[mm_addr_so.reg[mm_addr_select]] := mm_data_so[mm_data_select]
     []   main_output := mm_data_so[ mm_data_select ]
     od;
     mm_addr_so.reg[ mm_addr_select ] := mm_addr_so.reg[ mm_addr_select ] - 1;
until  mm_data_select = reg.inst_reg.c;
```

Figure 10 S*A Description

allocation of a scratch register is needed because the semantics of the **until** condition require a test for zero. The code is thus changed to what is shown in Figure 11b.

At this point the compactor indicates its inability to "pack" the **repeat** statement into one nano-word on the QM-1. (The body of the S*(QM-1) **repeat** statement must translate into one nano-word). The programmer decides to expand the macro in-line, in an attempt to locate unnecessary statements (Figure 11c). The statement

ks := mm_addr_select

sets up the input/output buses of the secondary QM-1 alu (the *index* alu [nano79]) to the register pointed to by **mm_addr_select** (the stack pointer). As this is an invariant expression in the loop (the stack pointer is decremented) it may be replaced by

ks := c_sp

a constant assignment, not executed in the loop body (set up before execution).

The compactor now generates a one nano-word implementation of the **repeat** statement capable of executing in seven T-periods [nano79] or 0.6 micro-seconds.

The importance of the tools discussed in this chapter can be characterized mainly by three things:

(a) Using the modularization constructs of the S*A description language enables one to conduct detailed analysis of isolated portions of the architecture, while retaining a

```
repeat                                                          11a
     mm_data_select := mm_data_select -1;
     MAIN_MEM_PUSH_G
until( mm_data_select == reg.inst_reg.c );
```

```
repeat                                                          11b
     mm_data_select := mm_data_select -1;
     MAIN_MEM_PUSH_G;
     f_scr1 := mm_data_select - reg.inst_reg.c
until( f_scr1 == 0 );
```

```
repeat                                                          11c
    mm_data_select := mm_data_select -1;

    main_memory[mm_addr_so.reg[mm_addr_select]] := mm_data_so[mm_data_select];
    ks := mm_addr_select;
    reg[ gspec.ks ] := xdecl reg[ gspec.ks ];

    f_scr1 := mm_data_select - reg.inst_reg.c
until  (f_scr1 == 0);
```

```
repeat                                                          11d
    mm_data_select := mm_data_select -1;
    main_memory[mm_addr_so.reg[mm_addr_select]] := mm_data_so[mm_data_select];

    ks := c_sp;
    reg[ gspec.ks ] := xdecl reg[ gspec.ks ];

    f_scr1 := mm_data_select - reg.inst_reg.c
until  (f_scr1 == 0);
```

Figure 11 S*(QM-1) Program Portion

global view of the overall system.

(b) An essentially top-down approach to both the design and implementation is possible. The design of certain aspects of the QM-C architecture could be postponed focusing attention on the more important portions of the design. Furthermore, the designer need not be concerned with the details and idiosyncrasies of the host machine, but can describe the target system at a more abstract level (for example, the QM-C multiplication and division instructions).

(c) Finally, as mentioned before and as detailed in [dasg81a], the close relationship between the description language and the implementation language made the transformation from the formal description to the final microcode representation very easy. The high level modularization of the S*A description is largely maintained in the S*(QM-1) code, easing the management and understandability of the code.

Even if programming in S*(QM-1) requires a detailed knowledge of the QM-1's architecture, expressing an algorithm using this language is orders of magnitude easier

than using a low level *nano-assembler* [nano79]. The predefined declarations of the QM-1 resources are a fixed part of the S*(QM-1) program and provide an excellent abstract representation of the machine; in fact most of the relevant architectural features of the QM-1 can be deduced from the semantics of the language together with these abstract declarations.

It should be noted, that programming in S*(QM-1) will probably always be an iterative process (similar to what is outlined above). The programmer first represents his algorithm at the highest possible level (very close to the S*A description) within the S*(QM-1) framework. Compiling (especially compacting) the code will reveal that certain portions of the code must be represented at a lower level, closer to the actual machine representation, in order to either achieve correct code, or to improve the quality of the generated machine representation.

Chapter 6

Conclusions

The design of a language-directed architecture for a microprogrammable machine has been examined. Of central importance is the premise that in order to design such an architecture, a compromise is needed between the "ideal" architecture for the language (as deduced from its characteristics and usage) and the "realities" of the host machine. A design method that disciplines this compromise is of utmost importance, such that neither the language nor the technology becomes the overly dominant factor. The design method outlined in previous chapters, and the usage of powerful new tools to realize the QM-C design, provided such a discipline, and should be considered the central theme of this work.

The S*A architectural description language was proven applicable to the design and description of a microprogrammed architecture. The primary advantage of the S*A language is that it enables the architect to describe the various design components at a *high level of abstraction*. That is, the description is relatively independent of the idiosyncrasies of a particular machine.

Many have been sceptical about the feasibility of high level microprogramming languages. The experience gained with the QM-C implementation in S*(QM-1) along with the work presented in [klas81,ride81], shows that defining a relatively large microprogrammed system is perfectly feasible using the S* - S*(QM-1) approach to high level microprogramming [dasg78,klas81a,ride81].

As was explained in previous chapters, the QM-C was designed, specifically, to be realized on the QM-1. The characteristics of the QM-1's nano-architecture placed several restrictions on the design. There are basically two facts that make the QM-1 less suitable as a host for language directed architectures:

1. *Addressability*. One of the most important characteristics of a good host architecture is its capability to address its memory in small chunks, preferably bits [hove78]. The QM-1 is word addressable, capable only of addressing 18-bit

words. The consequence is a restricted range of data types and storage classes.

2. *Decoding Flexibility.* In order to minimize the size of individual instructions, high flexibility must be possible when decoding the instruction parameters. The ability to extract variable sized fields from the instruction is highly desirable. The QM-1 was designed to interpret an instruction set, severly limited in format. As a consequence, a high loss of efficiency would be incurred if the an architecture implemented on the QM-1 deviated significantly from these limited formats.

Despite the above, this work has shown that a language-directed architecture can still be designed, using the QM-1 as a host. However, for this architecture to be practical, some of the features deemed desirable by the characteristics of the language had to be sacrificed for better utilization of the host architecture.

The evaluation of the performance of the QM-C architecture awaits the construction of a C compiler (based on the Portable C Compiler), and the completion of the S*(QM-1) compiler. However, preliminary studies do indicate the superiority of the QM-C to the MULTI machine. Programs written for the QM-C are smaller, and execute faster on the QM-C than if these same programs were compiled into MULTI. (See Appendix II for a comparison of the QM-C to MULTI).

6.1 Future Work

The design of the QM-C architecture was intended mainly as an exercise in language-directed machine design. Consequently, there are many features in the QM-1 architecture that are not utilized by the QM-C. One of the possible extensions to the work presented here, would be an investigation into how the C language might be augmented to directly support some of these hardware resources[12]. New data types would need to be added and language constructs provided to operate on them. (Example of these resources are the 6-bit register file, the RMI unit, main store accessing primitives, and other supervisory and auxiliary functions [nano79]). This would make it possible to truly microprogram the QM-1 in C, and perform general architectural emulations using the UNIX / QM-1 environment.

---

[12] The QM-C uses 118 of the 128 opcodes available. The remaining 10 could be used for this purpose.

Similar to the proposal in [leve79], a possible extension to the work on S*A and S*(QM-1), is the possibility of automatically generating portions of the code generation phase of the Portable C Compiler from an S*A description of the architecture and the S*(QM-1) definition of the instruction set. As mentioned in chapter three, the code generation phase of the Portable C Compiler is driven by a matching process that tries to match portions of the expression trees to prestored patterns (templates). When a match occurs, code is emitted. Some of these templates might be automatically generated from the S*A description, together with the S*(QM-1) definition, of individual instructions.

Two other interesting research problems are suggested by the work presented here:

(a) Examining the applicability and usage of architectural tuning techniques to the QM-C architecture. In particular, the methods proposed by Johnson [john79] and Sakamura [saka79], may be considered.

(b) Examining the possibility of automating the translation process from S*A architectural description to a microprogrammed implementation based on the S* schema [dasg81a].

With increased interest in distributed computer systems, and the introduction of general purpose microprogrammable *building blocks* [kral80], the possibility now exists of designing and implementing systems whose functionality is distributed among several processors. The system would consist of processors (of identical hardware) whose microprogrammed architecture would be dedicated to perform specific applications (language blocks, compiler blocks, editing blocks, etc.). Execution of tasks in such an environment would consist of a specific route through the system, and the compilers would be instrumented to make intelligent guesses about where in the system a particular program would run most efficiently, and generate code accordingly. (An extension of the compiler instrumentation presented in chapter three).

Interestingly, the software environment for such a system already exists: the UNIX operating system. Programmers in the UNIX environment solve their problems by combining together diverse tools, each designed to handle well defined portions of the problem. The UNIX tools may be categorised according to their primary function and each

would execute on a processor specially designed to efficiently execute this type of programs.

It is in the design and implementation of such distributed computer systems, that the methods and tools used in the QM-C design could be used to their full potential.

Bibliography

[aho75] Aho, A.V., S.C. Johnson, "Optimal Code Generation for Expression Trees", *ACM Journal* **23**(3), 1975, 488-501.

[alex75] Alexander, W.G., D.B. Wortman, "Static and Dynamic Characteristics of XPL programs", *Computer* **8**(11), 1975, 41-46.

[bash67] Bashkow, T.R., A. Sasson, A. Kronfield, "System Design of a FORTRAN Machine", *IEEE Trans. on Electronic Computers*, **Vol EC-16**, No 4, August 1967, 485-499.

[batt78] Battarel, G.J., R.J. Chevance, "Design of a High Level Language Machine", *Comp. Architecture News (SIGARC)* **6**(9), June 1978, 5-18.

[brin77] Brinch-Hansen, P., *The Architecture of Concurrent Programs*, Prentice Hall, N.J., 1977.

[burh78] Burkhard, W.A., R.D. Tuck, R.L. Hartung, *QM-1 MULTI Micro-programmer Guide*, Naval Surface Weapons Center, 1978, NSWC /DL TR -3834.

[burk78] Burke, H.J., A. Frick, Ch. Schlier, "High Level Language Oriented Hardware and the Post-Von Neuman era", *Proc. of the 5th Annual Symp. on Computer Architecture*, April 1978, 60-65.

[chu75] Chu, Y. *High Level Language Computer Architecture*, Academic Press, New York, 1975.

[chu75a] Chu, Y., "Concepts of High-Level Language Computer Architecture", *Proc. of the 1975 ACM Annual Conf.*, 6-13.

[coop80] Cooper, R.E.M., "The Direct Execution of Intermediate Languages on an Eclipse Computer", *SIGMICRO* **11**(1), March 1980.

[dasg78] Dasgupta, S., "Towards a Microprogramming Language Schema", *Proc. 11th Annual Workshop on Microprogramming* (MICRO 11), IEEE, N.Y., Nov. 1978, 144-153.

[dasg80] Dasgupta, S. "Some Aspects of High Level Microprogramming", *ACM Computing Surveys*, **12**(3), Sept. 1980, 295-324.

[dasg81] Dasgupta, S. "S*A : A Language for Describing Computer Architectures", *Proc. 5th Intl. Conf. on Computer Hardware Description Languages, and their Applications*, North Holland, Amsterdam, Sept. 1981.

[dasg81a] Dasgupta, S., M. Olafsson, "Towards a Family of Languages for the Design and Implementation of Machine Architectures", *Tech. Rept TR81-5*, Dept. of Computing Science, Univ. of Alberta, Edmonton, Canada, June 1981.

[dasg81b] Dasgupta, S., *Private Communications*, 1981.

[dasg82] Dasgupta, S. "Computer Design and Description Languages", in, M.C. Yovits (ed), *Advances in Computers*, **Vol. 21**, Academic Press, N.Y., 1982.

[demc76] Demco, J.C., T.A. Marsland, "An Insight into PDP-11-Emulation", *Proc. 9th Annual Workshop on Microprogramming*, Sept. 1976, 20-26.

[denn78] Denning, P.J., "A Question of Semantics", *Computer Architecture News (SIGARCH)* **6**(8), April 1978, 16-18.

[ditz80] Ditzel, D.R., "Program Measurements on a High-Level Language Computer", *IEEE Computer* **13**(8), August 1980.

[ditz80a] Ditzel, D.R., D.A. Patterson, "Retrospective on High-level Language Computer

Architecture", *Proc. 7th Annual Symp. on Computer Architecture*, May 1980, 97-104.

[elsh76] Elshoff, J.L., "A Numerical Profile of Commercial PL/I Programs", *Software - Practice and Experience* **6**(4), Oct.-Dec. 1976, 505-525.

[flaj79] Flajolet, P., J.C. Raoul, J. Vuillemin, "The Number of Registers Required for Evaluating Arithmetic Expressions", *Theoretical Comp. Sci.* **9**, July 1979, 99-125.

[flyn78] Flynn, M.J., "A Canonic Interpretive Program Form for Measuring "Ideal" HLL Architecture", *Computer Architecture News (SIGARCH)* **6**(8), April 1978, 6-15.

[flyn80] Flynn, M.J., "Directions and Issues in Architecture and Languages", *IEEE Computer*, October 1980, 5-22.

[gray81] Gray, C. *Private Communication*, June 1981.

[hagi80] Hagiwara, H., S. Tomita, S. Oyanagi, K. Shibayama, "A Dynamically Microprogrammable Computer with Low-Level Parallelism", *IEEE Trans. on Computers*, **Vol. C-29**, No. 7, July 1980, 577-594.

[hoev74] Hoevel, L.W., "Ideal Directly Executed Languages: An Analytical Argument for Emulation", *IEEE Trans. on Computers*, **C-23**, August 1974, 759-767.

[hove74a] Hoevel, L.W., "Languages for Direct Execution", *Proc. of ACM 7th Annual Workshop on Microprogramming*, 1974, 307-316.

[hove77] Hoevel, L.W., M.J. Flynn, "The Structure of Directly Executed Languages: A New Theory of Interpretive System Design", *Tech. Rept. 130*, Stanford University, March 1977.

[hove78] Hoevel, L.W., "Directly Executed Languages", Ph.D. Thesis, John Hopkins University, 1978.

[john75] Johnson, S.C. "Yacc - Yet Another Compiler-Compiler", *Comp. Sci. Tech. Rept. No.32*, Bell Laboratories, Murry Hill, N.J., 1975.

[john78] Johnson, S.C. "A Portable Compiler: Theory and Practice", *Proc. 5th ACM Symp. on Principles of Programming Languages*, Jan. 1978, 97-104.

[john79] Johnson, S.C., "A 32-bit Processor Design", *Comp. Sci. Tech. Rept. 80*, Bell Laboratories, Murry Hill, N.J., April 1979.

[john79a] Johnson, S.C., "A Tour Through The Portable C Compiler", Bell Laboratories, Murry Hill, N.J., 1979.

[jone80] Jones, J., "Towards a HLL Oriented Microprocessor Instruction Set", *Euromicro J.* **6**, 1980, 158-160.

[keed78] Keedy, J.L., "On the Evaluation of Expressions using Accumulators, Stacks and Store-to-Store Instructions", *Comp. Arch. News (SIGARC)* 7(4), Dec. 1978, 24-28.

[keed79] Keedy, J.L., "More on the Use of Stacks in the Evaluation of Expressions", *Comp. Arch. News (SIGARC)* 7(8), June 1979, 18-23.

[kern78] Kernigan, B.W., D. Ritchie, *The C Programming Language*, Prentice-Hall, N.J., 1978.

[klas81] Klassen, A., S. Dasgupta, "S*(QM-1) : An Instantiation of the High-Level Microprogramming Language Schema S* for the Nanodata QM-1", *Proc. 14th Annual Workshop on Microprogramming*, (MICRO 14), (submitted), IEEE, N.Y., Nov 1981. (Tech ept TR81-4, Dept. of Comp. Sci., Univ. of Alberta, Edmonton, Canada, June 1981)

[klas81a] Klassen, A., "S*(QM-1): An Experimental Evaluation of the High Level Microprogramming Language Schema S* using the Nanodata QM-1", M.Sc. Thesis, Univ. of Alberta, 1981.

[klas81b] Klassen, A., S. Dasgupta, "The Syntax and Semantics of the High Level Microprogramming Language S*(QM-1)", *Tech. Rept. TR81-3*, Dept. of Computing Science, Univ. of Alberta, Edmonton, Canada, June 1981.

[knut71] Knuth, D.E., "An Empirical Study of FORTRAN Programs", *Software - Practice and Experience* 1(2), 1971, 105-134.

[kral80] Kraley, M., R. Rettberg, P. Herman, R. Bessler, A. Lake, "Design of a User-Microprogrammable Building Block", *Proc. 13th Annual Workshop on Microprogramming*, Colorado, Dec. 1980, 106-114.

[laws68] Lawson, H.W. jr., "Programming Language Oriented Instruction Streams", *IEEE Trans. on Computers*, Vol C-17, No. 5, May 1968, 467-485.

[leff80] Leffler, S.J., "A Detailed Tour through the /6 Portable C Compiler", Dep. of Computer Engineering, Case-Western University, 1980.

[leve79] Leverett, B.W., et al., "An Overview of the Production Quality Compiler-Compiler Project", Carnegie-Mellon University, *Tech. Rept. No. 105*, 1979.

[lund77] Lunde, A. "Empirical Evaluation of Some Features of Instruction Set Processor Architecture", *Comm. ACM* 20(3), March 1977, 143-153.

[mead80] Mead, C.A., L. Conway, *An Introduction to VLSI Systems*, Addison-Wesley, M.A., 1980.

[myer77] Myers, G.J., "The Case Against Stack-Oriented Instruction Sets", *Comp. Arch. News (SIGARC)*, August 1977.

[myer78] Myers, G.J., "The Evaluation of Expressions in a Storage-to-Storage Architecture", *Comp. Architecture News (SIGARC)* **6**(9), June 1978, 20-24.

[myer78] Myers, G.J., *Advances in Computer Architecture*, John Wiley, 1978.

[nano79] Nanodata Computer Corporation, *The QM-1 Hardware Users Manual*, (Revised Edition), Buffalo, N.Y., 1979.

[neuh80] Neuhauser, C.J., "Analysis of the PDP-11 Instruction Stream", *Tech. Rept. 183*, Stanford University, Feb. 1980.

[olaf81] Olafsson, M., S. Dasgupta, "Syntax and Semantics of the Architectural Description Language S*A", *Tech. Rept. TR81-6*, Dept. of Computing Science, Univ. of Alberta, Edmonton, Canada, June 1981.

[orga78] Organic, E.F., J.A. Hinds, *Interpreting Machines : Architecture and Programming of the B1700/B1800 Series*, Elsevier, North-Holland, Inc., New York, 1978.

[patt80] Patterson, D.A., D.R. Ditzel, "The Case for the Reduced Instruction Set Computer", *Comp. Arch. News* **8**(6), Oct. 1980.

[patt81] Patterson, D.A., C.H. Sequin, "RISC I: A Reduced Instruction Set VLSI Computer", *Proc. of the 8th Annual Symp. on Computer Architecture*, May 1981, 443-457.

[raus76] Rauscher, T.G., A.K. Agrawala, "Developing Application Oriented Computer Architectures on General Purpose Microprogrammable Machines", *Proc. AFIPS 1976 Natl. Comp. Conf.* **45**, AFIPS Press, Motvale, N.J., 715-722.

[ride81] Rideout, D.J., "An Application of a Microcode Compaction Technique to the Nanodata QM-1 Nano-Architecture", M.Sc. Thesis, Univ. of Alberta, 1981.

[ride81b] Rideout, D.J. "Considerations for Local Compaction of Nanocode for the Nanodata QM-1", *Proc. 14th Annual Workshop on Microprogramming*, (MICRO 14), (submitted), IEEE, N.Y., Nov. 1981. (Tech. Rept. TR81-7, Dept. of Computing Sci., Uni of Alberta, Edmonton, Canada, June 1981.

[ritc78] Ritchie, D.M., K. Thompson, "The UNIX Time-Sharing System", *The Bell System Technical Journal*, vol 15, No. 6, July-August 1978, 1905-1930.

[rive79] Rivest, R.L., "The BLIZZARD Computer Architecture", *Comp. Arch. News (SIGARC)* 7(9), August 1979, 2-11.

[robi76] Robinson, S.K., I.S. Torsun, "An Empirical Analysis of FORTRAN Programs", *The Computer Journal* 19(1), 56-62.

[russ78] Russell, R.D., "The PDP-11: A Case Study of How Not to Design Condition Codes", *The 5th Annual Symp. on Computer Architecture*, April 1978, 190-194.

[saka79] Sakamura, K., T. Morokuma, H. Asso, H. Ilzuka, "Automatic Tuning of Computer Architectures", *Proc AFIPS 1979 Natl. Comp. Conf.*, 48, AFIPS Press, Montvale, N.J., 499-512.

[salv75] Salvadori, A., J. Gordon, C. Capstic, "Static Profile of COBOL Programs", *SIGPLAN Notices (ACM)* 10(8), 1975, 20-33.

[saun79] Saunders, S.E., "Compiling Customized Executable Representations and Interpreters", Carnegie-Mellon University, *Tech. Rept. No. 127*, June 1979, CMU-CS-79-127.

[schu73] Schutte, L.J., "A Report on the Value of some Advanced High-Level Language Operations", *Proc. of a Symp. on High-Level Language Computer Architecture*, Nov. 1973, 117-123.

[schu77] Schulthess, P.T., E.P. Mumprect, "Reply to the Case Against Stack-Oriented Instruction Sets", *Computer Architecture News (SIGARCH)* 6(5), Dec. 1977.

[seth75] Sethi, R., "Complete Register Allocation Problems", *SIAM Journal on Computing* **4**(3), Sept. 1975, 226-248.

[shap78] Shapiro, M.D., "The Criterion COBOL System", *Proc. AFIPS Natl. Comp. Conf.* **47**, 1978, 1049-1054.

[shus78] Shustec, L.J., "Analysis and Performance of Computer Instruction Sets", Ph.D. Dissertation, Stanford University, 1978.

[site78] Sites, R.L., "A Combined Register-Stack Architecture", *Computer Architecture News (SIGARCH)* **6**(8), April 1978, 19.

[site79] Sites, R.L., "Machine Independent Register Allocation", *Proc. SIGPLAN Symp. on Compiler Construction*, Denver, Colo. 1979, 221-225.

[stre78] Strecker, W.D. "VAX 11/780: A Virtual Address Extension to the DEC PDP-11 Family", *Proc. AFIPS 1978 Natl. Comp. Conf.*, **47**, AFIPS Press, Montvale, N.J., 1978, 967-980.

[sutp79] Sutphen, S., "Teaching and Research Experiences with an Emulation Laboratory", *Proc. AFIPS Natl. Comp. Conf.* **48**, 1979, 13-18.

[tane78] Tanenbaum, A.S., "Implication of Structured Programming for Machine Architecture", *Comm. ACM* **21**(3), March 1978, 237-246.

[wade72] Wade, B.W., V.B. Schneider, "The L-Machine: A Computer Instruction Set for Efficient Execution of High-Level Language Programs", *Record of the 5th Annual Workshop on Microprogramming*, New York, 1972, 81-82.

[wort72] Wortman, D.B., "A Study of Language Directed Computer Design", *CSRG-20*, University of Toronto, 1972.

Appendix I

The QM-C Instruction Set

Mnemonic	Operands	Function	Condition codes :	H	C	S	R	O	L
INCR	c6, r	r ← (r)+c6		-	*	*	*	*	-
INCM	c6,rb,o18	((rb)+o18) ← ((rb)+o18)+c6		-	*	*	*	*	-
INCG1	rb,o6	((rb)+o6) ← ((rb)+o6)+1		-	*	*	*	*	-
INCEC	c5,o6	((eb)+o6) ← ((eb)+o6)+c5		-	*	*	*	*	-
INCE1	o11	((eb)+o11) ← ((eb)+o11)+1		-	*	*	*	*	-
INCSC	c5,o6	((fp)+o6) ← ((fp)+o6)+c5		-	*	*	*	*	-
INCS1	o11	((fp)+o11) ← ((fp)+o11)+1		-	*	*	*	*	-
DECR	c6, r	r ← (r)-c6		-	*	*	*	*	-
DECM	c6,rb,o18	((rb)+o18) ← ((rb)+o18)-c6		-	*	*	*	*	-
DECG1	rb,o6	((rb)+o6) ← ((rb)+o6)-1		-	*	*	*	*	-
DECEC	c5,o6	((eb)+o6) ← ((eb)+o6)-c5		-	*	*	*	*	-
DECE1	o11	((eb)+o11) ← ((eb)+o11)-1		-	*	*	*	*	-
DECSC	c5,o6	((fp)+o6) ← ((fp)+o6)-c5		-	*	*	*	*	-
DECS1	o11	((fp)+o11) ← ((fp)+o11)-1		-	*	*	*	*	-
ADDRR	r1,r2	r2 ← (r2)+(r1)		-	*	*	*	*	-
ADDCR	c18,r,rd	rd ← (r)+o18		-	*	*	*	*	-
ADDMR	rb,o18,r	r ← (r)+((rb)+o18)		-	*	*	*	*	-
ADDER	o6,r	r ← (r)+((eb)+o6)		-	*	*	*	*	-
ADDSR	o6,r	r ← (r)+((fp)+o6)		-	*	*	*	*	-
ADDRM	r,rb,o18	((rb)+o18) ← ((rb)+o18)+(r)		-	*	*	*	*	-
ADDRE	r,o6	((eb)+o6) ← ((eb)+o6)+(r)		-	*	*	*	*	-
ADDRS	r,o6	((fp)+o6) ← ((fp)+o6)+(r)		-	*	*	*	*	-
ADDCM	c11,rb,o12	((rb)+o12) ← ((rb)+o12)+c11		-	*	*	*	*	-
ADDMM	rb,o6,rb,o12	((rb)+o12) ← ((rb)+o12)+((rb)+o6)		-	*	*	*	*	-
ADDEM	o11,rb,o12	((eb)+o11) ← ((eb)+o11)+((rb)+o12)		-	*	*	*	*	-
ADDSM	o11,rb,o12	((fp)+o11) ← ((fp)+o11)+((rb)+o12)		-	*	*	*	*	-
SUBRR	r1,r2	r2 ← (r2)-(r1)		-	*	*	*	*	-
SUBCR	c18,r,rd	rd ← (r)-o18		-	*	*	*	*	-
SUBMR	rb,o18,r	r ← (r)-((rb)+o18)		-	*	*	*	*	-
SUBER	o6,r	r ← (r)-((eb)+o6)		-	*	*	*	*	-
SUBSR	o6,r	r ← (r)-((fp)+o6)		-	*	*	*	*	-
SUBRM	r,rb,o18	((rb)+o18) ← ((rb)+o18)-(r)		-	*	*	*	*	-
SUBRE	r,o6	((eb)+o6) ← ((eb)+o6)-(r)		-	*	*	*	*	-
SUBRS	r,o6	((fp)+o6) ← ((fp)+o6)-(r)		-	*	*	*	*	-
SUBCM	c11,rb,o12	((rb)+o12) ← ((rb)+o12)-c11		-	*	*	*	*	-
SUBMM	rb,o6,rb,o12	((rb)+o12) ← ((rb)+o12)-((rb)+o6)		-	*	*	*	*	-
SUBEM	o11,rb,o12	((eb)+o11) ← ((eb)+o11)-((rb)+o12)		-	*	*	*	*	-
SUBSM	o11,rb,o12	((fp)+o11) ← ((fp)+o11)-((rb)+o12)		-	*	*	*	*	-
ANDRR	r1,r2	r2 ← (r2) & (r1)		-	*	*	*	*	-
ANDCR	c18,r,rd	rd ← (r) & o18		-	*	*	*	*	-
ANDMR	rb,o18,r	r ← (r) & ((rb)+o18)		-	*	*	*	*	-
ANDER	o6,r	r ← (r) & ((eb)+o6)		-	*	*	*	*	-
ANDSR	o6,r	r ← (r) & ((fp)+o6)		-	*	*	*	*	-
ANDRM	r,rb,o18	((rb)+o18) ← ((rb)+o18) & (r)		-	*	*	*	*	-
ANDRE	r,o6	((eb)+o6) ← ((eb)+o6) & (r)		-	*	*	*	*	-
ANDRS	r,o6	((fp)+o6) ← ((fp)+o6) & (r)		-	*	*	*	*	-
ANDCM	c18,rb,o6	((rb)+o6) ← ((rb)+o6) & c18		-	*	*	*	*	-
ANDCE	c18,o11	((eb)+o11) ← ((eb)+o11) & c18		-	*	*	*	*	-
ANDCS	c18,o11	((fp)+o11) ← ((fp)+o11) & c18		-	*	*	*	*	-
ANDMM	rb,o6,rb,o12	((rb)+o12) ← ((rb)+o12) & ((rb)+o6)		-	*	*	*	*	-
ANDEM	o11,rb,o12	((eb)+o11) ← ((eb)+o11) & ((rb)+o12)		-	*	*	*	*	-
ANDSM	o11,rb,o12	((fp)+o11) ← ((fp)+o11) & ((rb)+o12)		-	*	*	*	*	-

Mnemonic	Operands	Function	Condition codes :	H	C	S	R	O	L
IORRR	r1,r2	r2 ← (r2) | (r1)		-	*	*	*	*	-
IORCR	c18,r,rd	rd ← (r) | o18		-	*	*	*	*	-
IORMR	rb,o18,r	r ← (r) | ((rb)+o18)		-	*	*	*	*	-
IORER	o6,r	r ← (r) | ((eb)+o6)		-	*	*	*	*	-
IORSR	o6,r	r ← (r) | ((fp)+o6)		-	*	*	*	*	-
IORRM	r,rb,o18	((rb)+o18) ← ((rb)+o18) | (r)		-	*	*	*	*	-
IORRE	r,o6	((eb)+o6) ← ((eb)+o6) | (r)		-	*	*	*	*	-
IORRS	r,o6	((fp)+o6) ← ((fp)+o6) | (r)		-	*	*	*	*	-
IORCM	c18,rb,o6	((rb)+o6) ← ((rb)+o6) | c18		-	*	*	*	*	-
IORCE	c18,o11	((eb)+o11) ← ((eb)+o11) | c18		-	*	*	*	*	-
IORCS	c18,o11	((fp)+o11) ← ((fp)+o11) | c18		-	*	*	*	*	-
IORMM	rb,o6,rb,o12	((rb)+o12) ← ((rb)+o12) | ((rb)+o6)		-	*	*	*	*	-
IOREM	o11,rb,o12	((eb)+o11) ← ((eb)+o11) | ((rb)+o12)		-	*	*	*	*	-
IORSM	o11,rb,o12	((fp)+o11) ← ((fp)+o11) | ((rb)+o12)		-	*	*	*	*	-
RSHRR	r1,r2	r2 ← (r2) >> (r1)		*	*	*	*	*	*
RSHCR	c6,r	r ← (r) >> c6		*	*	*	*	*	*
RSHRM	r,rb,o18	((rb)+o18) ← ((rb)+o18) >> (r)		*	*	*	*	*	*
RSHCM	c6,rb,o18	((rb)+o18) ← ((rb)+o18) >> c6		*	*	*	*	*	*
LSHRR	r1,r2	r2 ← (r2) << (r1)		*	*	*	*	*	*
LSHCR	c6,r	r ← (r) << c6		*	*	*	*	*	*
LSHRM	r,rb,o18	((rb)+o18) ← ((rb)+o18) << (r)		*	*	*	*	*	*
LSHCM	c6,rb,o18	((rb)+o18) ← ((rb)+o18) << c6		*	*	*	*	*	*
MULRR	r1,r2	r2 ← (r2) * (r1)		-	*	*	*	*	-
DIVRR	r1,r2	r2 ← (r2) / (r1) r1 ← mod((r2),(r1))		-	*	*	*	*	-
ALURR	r1,r2,rd,c6	rd ← (r2) c6 (r1) operation given by c6		-	*	*	*	*	-
CMPRR	r1,r2	(r2) - (r1)		-	*	*	*	*	-
CMPCR	c6,r	(r) - c6		-	*	*	*	*	-
CMPMR	rb,o18,r	(r) - ((rb)+o18)		-	*	*	*	*	-
CMPER	o6,r	(r) - ((eb)+o6)		-	*	*	*	*	-
CMPSR	o6,r	(r) - ((fp)+o6)		-	*	*	*	*	-
CMPCM	c11,rb,o12	((rb)+o12) - c11		-	*	*	*	*	-
CMPMM	rb,o6,rb,o12	((rb)+o12) - ((rb)+o6)		-	*	*	*	*	-
JMPEQ	o11	pc ← (pc) + o11 if zero pc ← (pc) + 1 if not zero		-	-	-	-	-	-
JMPNE	o11	pc ← (pc) + o11 if not zero pc ← (pc) + 1 if zero		-	-	-	-	-	-
JMPGE	o11	pc ← (pc) + o11 if non-negative pc ← (pc) + 1 if positive		-	-	-	-	-	-
JMPLT	o11	pc ← (pc) + o11 if negative pc ← (pc) + 1 if non-negative		-	-	-	-	-	-
JMPR	o11	pc ← (pc) + o11		-	-	-	-	-	-
JMPA	o11	pc ← (eb) + o11		-	-	-	-	-	-
JMPAL	o18	pc ← (eb) + o18		-	-	-	-	-	-
SWITCH	r,c6	pc ← (pc) + (r) & c6		-	-	-	-	-	-
CALL	o11	pc ← (eb)+o11+1 save registers, allocate stack		-	-	-	-	-	-
CALLA	o18	pc ← (eb)+o18+1 save registers, allocate stack		-	-	-	-	-	-

Mnemonic	Operands	Function	Condition codes :	H	C	S	R	O	L
RETURN	r,c6	restore registers, deallocate stack, (pc) ← (old pc) + 1		-	-	-	-	-	-
MOVRR	r1,r2	r2 ← (r1)		-	-	-	-	-	-
MOVCR	c18,r	r ← c18		-	-	-	-	-	-
MOVSCR	c6,r	r ← c6		-	-	-	-	-	-
MOVMR	rb,o18,r	r ← ((rb)+o18)		-	-	-	-	-	-
MOVER	o6,r	r ← ((eb)+o6)		-	-	-	-	-	-
MOVSR	o6,r	r ← ((fp)+o6)		-	-	-	-	-	-
MOVRM	r,rb,o18	((rb)+o18) ← (r)		-	-	-	-	-	-
MOVRE	r,o6	((eb)+o6) ← (r)		-	-	-	-	-	-
MOVRS	r,o6	((fp)+o6) ← (r)		-	-	-	-	-	-
MOVCM	c11,rb,o12	((rb)+o12) ← c11		-	-	-	-	-	-
MOVCE	c5,o6	((eb)+o6) ← c5		-	-	-	-	-	-
MOVCS	c5,o6	((fp)+o6) ← c5		-	-	-	-	-	-
MOVMM	rb,o6,rb,o12	((rb)+o12) ← ((rb)+o6)		-	-	-	-	-	-
MOVPR	rb,o18,r	r ← (rb)+o18		-	-	-	-	-	-
MOVPM	rb,o6,rb,o12	((rb)+o12) ← (rb)+o6		-	-	-	-	-	-
PUSHR		(sp) ← (r) sp ← (sp) - 1		-	-	-	-	-	-
PUSHM	rb,o6	((sp)) ← ((rb)+o6) sp ← (sp) - 1		-	-	-	-	-	-
PUSHE	o11	((sp)) ← ((eb)+o11) sp ← (sp) - 1		-	-	-	-	-	-
PUSHS	o11	((sp)) ← ((fp)+o11) sp ← (sp) - 1		-	-	-	-	-	-
PUSHC	c11	((sp)) ← c11 sp ← (sp) - 1		-	-	-	-	-	-
PUSHP	rb,o6	((sp)) ← (rb)+o6 sp ← (sp) - 1		-	-	-	-	-	-
PUSHPE	o11	((sp)) ← (eb) + o11 sp ← (sp) - 1		-	-	-	-	-	-
PUSHPS	o11	((sp)) ← (fp) + o11 sp ← (sp) - 1		-	-	-	-	-	-

Legend :

() - contents of
rb - genral base register
eb - external base register
sp - stack pointer

r - register
fp - frame pointer
pc - program counter
rd - destination register (for three operand instructions)

cN - N-bit constant

oN - N-bit offset

Condition codes :
H - Shifter high bit
S - Sign
O - Overflow

C - Carry
R - Result
L - Shifter low bit

* : affected

- : unaffected

Appendix II

QM-C vs MULTI :

A Comparison

Complete evaluation of the performance of the QM-C architecture awaits the construction of a compiler (from the Portable C compiler) and the completion of the S*(QM-1) compiler [klas81a]. Once this is done, programs may be compiled and the generated code analysed both dynamically and statically [lund77, neuh80]. However, for the purposes of illustration, C programs can be hand-compiled into the QM-C instruction set and compared to the same programs on other architectures. This comparison is primarily in terms of number of instructions emitted and the size of the executable image, but can give some idea of the efficiency of the QM-C architecture in general and specifically provide a basis for comparison to the MULTI machine [burh78], which the QM-C is intended to replace.

The difficulty in comparing the QM-C to the MULTI machine lies in the basic difference between the two instruction sets. The MULTI instruction set is designed entirely as a machine oriented instruction set with no intention to support the execution of any high level language. MULTI provides no mechanism to support dynamic storage and a very elementary support for procedure call and return. On the other hand, a substantial portion of the MULTI instruction set is reserved for supporting the various QM-1 resources. These instructions would not be generated by a compiler compiling conventional high level languages into MULTI. (For example, I/O instructions, instructions to access QM-1's main memory and various supervisory and auxiliary functions.) Such instructions will be needed to support the QM-C architecture on the QM-1, but, likewise, will not be generated by a C compiler for the QM-C and are therefore not considered parts of its external architecture.

To gain some insight into the actual code a compiler would generate for the QM-C, a small C program (see Figure 12) was constructed that might be considered representative of a typical application of the QM-C. This program decodes an instruction for a hypothetical machine. It isolates the opcode and on the basis of a format code, either prepares to zero an accumulator or perform an operation on a register and an immediate operand. Some checks are performed and profiling information collected. This algorithm is purely hypothetical and of no consequence.

```
int     acc;

struct  form {
        char    opc;
        int     op1;
        int     op2;
        } buf;

int     reg[7];

decod( opform, oprnd_a, oprnd_b )
int     opform, oprnd_a, oprnd_b;
{
        register int tform,status;
        int          count;

        count = tal();
        buf.opc = opform >> 8;
        tform = opform & 03;

        switch (tform) {

                case 1: buf.op1 = acc;
                        buf.op2 = 0;
                        status = check(1);
                        break;

                case 2: buf.op1 = reg[oprnd_a];
                        buf.op2 = oprnd_b;
                        status = check(2);

                }

        if(( status == 0 ) && ( count < 4 ))
                return(0);
        else
                return(1);
}
```

Figure 12 C Example Program

Shown as Figure 13 is the same program hand-compiled into the QM-C instruction set. This implementation requires 29 instructions, occupying 39 words of storage or 702 bits. With the mask word the total program space is 720 bits. The data space is 11 words static, and a stack frame of 8 words or a total of 342 bits.

This same program was coded in MULTI [gray81] and is shown in Figure 14. Since MULTI lacks stack management capabilities, a different calling mechanism must be used. The routines "csav" and "cret" are assumed to handle saving and restoring of local variables. Also, a parameter area must be set up within each routine to store parameters to called procedures.

The MULTI implementation requires 42 instructions or a program space of 61 words (1098 bits) and a static data space of 17 words. The actual implementation of the

```
_acc:       .blk      1,0
_buf:       .blk      3,0
_reg:       .blk      7,0

_decod:     RM1*4096.+SS1                ;Mask word

            call      _tal
            movrs     t0,0               ;result to 'count'
            movsr     FS1+2,t0           ;prepare to shift 'opform'
            rshcr     9.,t0              ;shift right 9 places
            movre     t0,_buf            ;result to 'buf.opc'

            movsr     FS1+2,t0           ;repare to and
            andcr     3.,t0,v7           ;and into 'tform'

            switch    v7,3               ;table 'switch' implement
            L1
            L2
            L3
            L1
L2:         movmm     eb,_acc,eb,_buf+1  ;assign 'acc' to 'buf.op1'
            movce     0,_buf+2           ;zero 'buf.op2'
            pushc     1                  ;push parameter to 'check'
            call      _check
            movrr     t0,v6              ;result into 'status'
            jmpr      L1                 ;relative branch

L3:         movpr     eb,_reg,t0         ;start of 'reg' to register
            addsr     FS1+1,t0           ;index into 'reg' with 'oprnd_a'
            movmm     t0,0,eb,_buf+1     ;'reg[oprnd_a]' to 'buf.op1'
            movmm     fp,FS1,eb,_buf+2   ;'oprnd_b' to 'buf.op2'
            pushc     2
            call      _check
            movrr     t0,v6              ;result to 'status'

L1:         cmpcr     0,v6               ;'status == 0' ?
            jmpne     L4
            cmpcs     4,0                ;'count < 4' ?
            jmpge     L4
            movscr    0,t0               ;'return(0)'
            return    RM1,ARGS1

L4:         movscr    1,t0               ;'return(1)'
            return    RM1,ARGS1

            RM1       =       v6       ;lowest register used
            FS1       =       v7-RM1+4 ;distance to parameters on stack
            SS1       =       1        ;size of area for automatics on stack
            ARGS1     =       3        ;size of area for parameters on stack
```

Figure 13 QM-C Code Example

"csav" and "cret" routines determines the size requirements of save areas and is not shown here.

Thus, in size the QM-C exhibits a substantial improvement over the MULTI implementation. Even without taking the "csav" and "cret" routines into account, the QM-C code is approximately 34% smaller. More interesting would be to compare the execution times of these two representations, but since the timing of individual QM-C instructions

is still unknown (depends on the quality of the compaction phase of the S*(QM-1) compiler), very little can be said about the relative execution times. A rough estimate for the "CALL" instruction on the QM-C found that it would probably be executable in 2.6 micro-seconds plus 0.54 micro-seconds for each register saved [ride81]. The "csav" and "cret" routines have not been microcoded for MULTI, but similar routines have been coded and timed as a part of an experiment with the PDP-11 emulator [demc76,sutp79]. The times obtained were 11.84 micro-seconds for "csav" and 14 for "cret". Although these times are only rough estimates of the times for the MULTI versions of "csav" and "cret", procedure calls on the QM-C using the "CALL" and "RETURN" instructions should be considerably faster.

To summarize, even if complete evaluation of the static and dynamic characteristics of the QM-C architecture awaits "porting" the Portable C Compiler to the QM-C and the completion of the S*(QM-1) compiler, the preliminary results presented above seem to indicate the feasibility of replacing the MULTI machine with the QM-C.

Figure 14 MULTI Implementation

```
acc:    .blk    1,0
buf:    .blk    3,0
reg:    .blk    7,0

decod:  baln    r.ret,r.zero,csav           ;save context

          .blk  1,3                         ;# of words for parameters
          .blk  1,1                         ;# of words for locals
opform:   .blk  1,0
oprnd_a:  .blk 1,0
tform   =       1
status  =       2
count     .blk  1,0
parbuf:   .blk  1,0                         ;buffer for parameters
        ldn     r.pars,r.zero,parbuf-1

        baln    r.ret,r.zero,tal            ;'count = tal()'
        std     r.result,r.zero,count

        ldn     3,r.zero,opform             ;'buf.opc = opform >> 9'
        srai    3,9
        std     3,r.zero,buf+0

        ldn     3,r.zero,opform             ;'tfrom = opform & 03'
        ann     3,3,03
        movr    3,tform

        movr    tform,3                     ;'switch (tform)'
        sbi     3,1
        bnz     3,L1

        ldn     3,r.zero,acc                ;'case 1 :'
        std     3,r.zero,buf+1

        ldi     3,0
        std     3,r.zero,buf+2

        ldi     3,1
        sty     3,1
        baln    r.ret,r.zero,check
        sbi     r.pars,1
        movr    r.result,status

        b       L2

L1:     ldn     3,r.zero,oprnd_a            ;'case 2 :'
        adi     3,reg
        ld      3,3
        std     3,r.zero,buf+1

        ldn     3,r.zero,oprnd_b
        std     3,r_zero,buf+2

        ldi     3,2
        sty     3,1
        baln    r.ret,r.zero,check
        sbi     r.pars,1
        movr    r.result,status

L2:     bnz     status,L3                   ;'status == 0' ?

        movr    status,3                    ;'status < 4' ?
        sbi     3,4
        bpl     3,L3

        ldi     result,0                    ;'return(0)'
        baln    r.par,r.zero,cret

L3:     ldi     result,1                    ;'return(1)'
        baln    r.par,r.zero,cret
```

Appendix III

Excerpts from the S*A description of the QM-C architecture

QM-C Architecture -- Overview

```
sys QM_C;

    sys INSTRUCTION_CYCLE;

        mech INSTRUCTION_FETCH;   ...  endmech;
        mech INSTRUCTION_DECODE;  ...  endmech;

        sys INSTRUCTION_EXEC;

            mech FETCH_OPRND;     ...  endmech;
            mech EXECUTE;         ...  endmech;
            mech DEPOSIT_OPRND;   ...  endmech;

        endsys;

    endsys;

    sys MEMORY;

        mech MAIN_MEM;   ...  endmech;
        mech AUX_MEM;    ...  endmech;

    endsys;

    sys INTERRUPT;

        mech INT_HANDLERS;   ...  endmech;
        mech STATE_CHANGE;   ...  endmech;

    endsys;

    sys NANO_ARCHITECTURE;

        mech EXEC_PRIM;   ...  endmech;
        mech IO_PRIM;     ...  endmech;
        mech MEM_PRIM;    ...  endmech;

    endsys;

endsys;
```

```
sys QM_C;
    /*
     *
     *  QM-C  Architectural Definition.
     *
     *        The QM-C architecture is defined here as it is
     *        implemented on the QM-1. In QM-1 terminology the
     *        QM-C is a micro-machine (machine executing out of
     *        control store).
     *
     *        The QM-C consists of four subsystems :
     *
     *        INSTRUCTION_CYCLE :  Defines the instruction fetch,
     *                             decode and execute mechanisms.
     *
     *        MEMORY            :  Defines the mechanisms to access
     *                             the QM-C memories. Main memory
     *                             (QM-1's control store) and auxillary
     *                             memory (QM-1's main store).
     *
     *        INTERRUPT         :  Defines the QM-C interrupt mechanism.
     *
     *        NANO_ARCHITECTURE :  Contains mechanisms which are
     *                             considered atomic at this
     *                             description level.
     *
     *
     */

    /*
     *   Type declarations of global types.
     *
     */
    type    ls_register      = seq [ 17..0 ] bit;
    type    es_register      = seq [ 17..0 ] bit;
    type    f_register       = seq [ 5..0 ] bit;
    type    bus              = seq [ 17..0 '] bit;

    type    k_vector_register = seq [ 5..0 ] bit;
    type    branch_register   = seq [ 9..0 ] bit;

    /*
     *   Global variables, shared by all QM-C subsystems. These variables
     *   represent actual QM-1 hardware resources named for consistency
     *   with S*(QM-1). In the QM-C architecture they are used under synonyms.
     *
     */

Not shown here .....
```

```
/*
 * QM-C global resources - Synonyms :
 *
 */
  syn  mm_addr_so  :  tuple                                  /* Main mem address sources  */
                        reg     : array [ 0..31 ] of ls_register
                                  with mm_addr_select
                                : bus
                        index   : array [ 0..3 ]  of ls_register
                                  with mm_index_select
                                : bus
                      endtup    =    control_store_address_source;

  syn  aux_dest    :  array [ 0..39 ] of ls_register     /* Aux. memory destination  */
                      with aux_out_select

                   :  tuple
                           reg    : array [ 0..31 ] of ls_register
                           port   : array [ 0..7 ]  of es_register
                      endtup  =  main_store_destination;

  syn  aux_source  :  array [ 0..39 ] of seq [ 17..0 ] bit /* Aux memory address/data source */
                      with aux_in_select

                   :  tuple
                           reg    : array [ 0..31 ] of ls_register
                           port   : array [ 0..7 ]  of es_register
                      endtup  =  main_store_source;

  syn  aux_output  =  main_store_output;                 /* Aux. memory output bus  */
  syn  main_output =  control_store_output;              /* Main memory output bus  */

  syn  instruction_address  =  kn;
  syn  alu_operation        =  kalc;

  syn  reg         : array [ 0..31 ] of ls_register     /* QM-C register file  */
                     with  dest, source, addr, alt_source,
                           reg.inst_reg.ab.a, reg.inst_reg.ab.b

                   : tuple
                           *       : array [ 0..11 ] of ls_register  /*  Not used by QM-C  */
                         temp      : array [ 0..3 ] of ls_register   /*  Temprarys         */
                         var       : array [ 0..7 ] of ls_register   /*  Variable registers  */
                         index     : array [ 0..3 ] of ls_register
                                     with mm_index_select
                                   : tuple
                                         fp  :  ls_register          /* Frame pointer    */
                                         pc  :  ls_register          /* Program counter  */
```

```
                                eb  :  ls_register        /* External base    */
                                ax  :  ls_register        /* Aux mem index    */
                              endtup
                    sp      : ls_register                 /* Stack pointer      */
                    scr1    : ls_register                 /* Nano scratch reg #1*/
                    scr2    : ls_register                 /* Nano scratch reg #2*/
                   inst_reg : tuple                       /* Instruction register */
                                opcode : seq [ 6..0 ]  bit
                                ab     : tuple
                                           a  : seq [ 4..0 ] bit
                                           b  : seq [ 5..0 ] bit
                                         endtup
                              endtup
                              : tuple                     /* 6 // 18 interface */
                                c  :  f_register
                                a  :  f_register
                                b  :  f_register
                              endtup
                  endtup  = local_store;

syn  mm_addr_select : seq [ 5..0 ] bit          /* Selects main memory address source  */
                    = f_store.fcia[ 5..0 ];


syn  mm_index_select: seq [ 1..0 ] bit          /* Selects  address+offset source */
                    = f_store.fmpc[ 1..0 ];

syn  mm_data_source : array [ 0..31 ] of ls_register /* Main memory data sources  */
                      with  mm_data_select
                    = reg;

syn  mm_data_select : seq [ 5..0 ] bit          /* Selects mm data source  */
                    = f_store.fcid[ 5..0 ];

syn  aux_in_select  : f_register                /* Selects data/address in  */
                    = f_store.fmix;
syn  aux_out_select : f_register                /* Selects  destination reg  */
                    = f_store.fmod;
/*
 *
 * QM-C global constants :
 *
 */
 const  n_pc   : dec(6) 25;
 const  n_sp   : dec(6) 28;
 const  n_fp   : dec(6) 24;
 const  n_eb   : dec(6) 26;

 const  n_scr1 : dec(6) 29;
```

```
const  n_scr2   : dec(6) 30;

const  n_inst_reg     : dec(6) 31;
/*
 *  Special constants (implementation dependent)
 *
 */
const  mm_base : oct(6) ..;                    /* Main mem base page  */
const  system_base : oct(6) ..;                /* Main mem base page of syst. externals */
const  clock_addr : oct(6) ..;
const  ii8_addr    : oct(6) ..;
const  ii9_addr    : oct(6) ..;
const  ch3_addr    : oct(6) ..;
const  ch4_addr    : oct(6) ..;

sys INSTRUCTION_CYCLE;
    /*
     *  Definition of the QM-C instruction pipeline.
     *
     *      Subsystems  :      INSTRUCTION_EXEC
     *
     *      Mechanisims :      INSTRUCTION_FETCH
     *                         INSTRUCTION_DECODE
     *
     */
    mech INSTRUCTION_FETCH;
        /*
         *  Context of Activation :  At end of each instruction and internal
         *                           interrupt handlers.
         *
         *     As instruction i activates FETCH, decoding of instruction i+1
         *     is initiated concurrent with fetch of instruction i+2.
         *
         *  Public processes   :  NEXT   - Start pipeline.
         *
         *  Private processes  :  None.
         *
         */
        proc NEXT;

          reg.index.pc := reg.index.pc+1;
          act MAIN_MEM.READ_P1 ▯ act INSTRUCTION_DECODE.SELECT_NEXT;

        endproc;

    endmech;

    mech INSTRUCTION_DECODE;
      /*
```

```
 *  Context of Activation:  Instruction fetch - to decode instruction
 *                          fetch by previous activation of fetch mechanism.
 *
 *     This is a complete description of the QM-C instruction decode
 *     but does not completely describe the underlying QM-1 priority
 *     select mechanism.
 *
 *
 *  Public processes  :  SELECT_NEXT  - Select next nanoword to gate into
 *                                      execution matrix.
 *
 *  Private Processes :  None.
 *
 */
type int_addr_array = array [ 0..30 ] of tuple /* Map interrupt addresses out of E-store */

                                          page : seq [ 1..0 ] bit
                                          addr : seq [ 3..0 ] bit

                                      endtup ;

glovar active,pending : array [ .. ] of bit;   /* Interrupt flags */

privar operand_buf    : seq [ 10..0 ] bit;     /* Operand receive register */
privar i   : seq [ .. ] bit;                   /* Counter */

syn   int_addr  : int_addr_array               /* Interrupt addresses */
                = external_store.interrupt_address;
syn   nano_page : seq [ 2..0 ] bit             /* One of four nanostroe pages */
                = f_store.fidx[ 2..0 ];

chan   allow_nano, allow_micro : bit;          /* Enable interrupt flags from current K */

proc SELECT_NEXT ;
  /*
   * Select next nano store entry address.
   *
   *          Prepares to execute next instruction, then
   *          searches interrupt levels for active interrupt.
   *          If found, redefine entry address to  point to handler,
   *          execute and exit.
   *          Otherwise execute next instruction.
   *
   *  Processes called :
   *
   *          EXEC_NPC - Nano primitive to execute nanoword
   *                     addressed by "inst_addr".
   *
   */
```

```
    /*
     * Form address of instruction routine and
     * save operands.
     *
     */
    do  instruction_address[ 9..7 ] := nano_page
     [] instruction_address[ 6..0 ] := main_output[ 17..11 ]
     [] operand_buf := main_output[ 10..0 ]
    od ;

    /*
     * Search for pending interrupts
     *
     */
    repeat
      if i < 12 => active[ i ] := pending[ i ] ^ allow_nano = 1
      || else   => active[ i ] := pending[ i ] ^ allow_micro = 1
      fi;

      /*
       * If interrupt level i active, set up nano address of handler
       *
       */
      if active[ i ] = 1  => do  instruction_address[ 9 ]    := 0
                              [] instruction_address[ 8..7 ] := int_address[ i ].page
                              [] instruction_address[ 6..4 ] := 0
                              [] instruction_address[ 3..0 ] := int_address[ i ].addr
                             od;
                             act EXEC_PRIM.EXEC_NPC [] exit;
      fi;

      i := i+1;
      until  i = 32;

    /*
     * No interrupt taken - execute next instruction
     *
     */
    do  reg.inst_reg[ 17..11 ] := 0
     [] reg.inst_reg[ 10..0 ]  := operand_buf
    od;

    act EXEC_PRIM.EXEC_NPC;

  endproc;

endmech;
```

```
sys  INSTRUCTION_EXEC;

     /*
      *  This system controls instruction execution. The primary
      *  mechanism in this system is EXECUTE, activated by the
      *  primitive function EXEC_NPC. The two other mechanisms are
      *  semi-private mechanisms e.i. they are only activated by
      *  processes in EXECUTE.
      *
      *  Subsystems   : None
      *
      *  Mechanisms   : FETCH_OPRNDS
      *                 EXECUTE
      *                 DEPOSIT_OPRNDS
      *
      */
     syn  cond_cod  :  tuple                    /* Condition codes   */

                              shift_high : bit
                              carry      : bit
                              sign       : bit
                              result     : bit
                              overflow   : bit
                              shift_low  : bit

                       endtup
                     = f_store.fist;

     syn  dest       : seq [ 5..0 ] bit         /* Destination register index */
                     = f_store.faod [ 5..0 ];
     syn  source     : seq [ 5..0 ] bit         /* Source register index */
                     = f_store.fail [ 5..0 ];
     syn  alt_source : seq [ 5..0 ] bit         /* Alternate source index */
                     = f_store.fair [ 5..0 ];
     syn  addr       = mm_addr_select;          /* Destination address pointer */


     chan  carry_out_hold;                      /* Carry out of alu  */
     chan  shift_overflow;                      /* Overflow from shifter */
     chan  alu_overflow;                        /* Overflow from alu */

     mech  FETCH_OPRNDS;

          /*
           *    Context of activation  :  Intruction execution to calculate effective
           *                              address of operands and fetch from memory.
           *
           */
```

```
    Not shown ....

endmech;

mech  EXECUTE;

    /*
     *  Context of Activation :  To execute instructions. Opcode
     *                           decoded by INSTRUCTION_DECODE.
     *                           The global processes in this mechanism
     *                           are activated from nano-primitive EXEC.
     *
     *  Only two processes shown as examples :
     *
     *  INCM  -  Increment memory variable
     *  CALL  -  Procedure call
     */

    proc INCM;

        /*
         *  Increment memory
         *
         *    Format   :  incm    r, c6, o18
         *
         *    Function :  ((r)+o18) <- ((r)+o18) + c6
         *
         */

        call FETCH_OPRND.FORM_2;                    /* Fetch operands        */

        reg[ dest ] := reg[ dest ] + reg[ source ];
        call SET_STATUS;                            /* Set condition codes   */

        call DEPOSIT_OPRND.REG;                     /* Write to mem. and fetch next */

    endproc;

    proc CALL;

        /*
         *  Procedure call.  First word of each procedure is a mask word indicating
         *                   the number of registers used (thus have to be saved)
         *                   and size of stack to be allocated for automatics and
         *                   temporaries.
         *
         *  Format  :  call  o11
         *
```

```
        *  Function : save registers 'pc' through 'reg[ ((eb)+o11)[ 17..11 ] ]' on stack
        *                (fp) <- (sp)
        *                (sp) <- (sp) - ((eb)+o11)[ 11..0 ]
        *                (pc) <- (eb)+o11+1
        *
        */

        do   reg.index.pc := reg.index.pc+1        /* Point reg.index.pc to next instruction */
         []  reg.scr1 := reg.inst_reg              /* Save offset                            */
         []  mm_index_select := n_eb
        od;

        call MAIN_MEM.READ_I_L;                    /* Read mask word                         */
        reg.inst_reg := main_output;               /* Prepare to decode                      */

        do   mm_addr_select := n_sp                /* Prepare to save reg's                  */
         []  mm_data_select := n_pc+1
        od;

        repeat                                     /* Save registers                         */
           mm_data_select := mm_data_select - 1;
           call MAIN_MEM.PUSH_G;
        until  mm_data_select = reg.inst_reg.c;

        reg.index.fp := reg.sp;                    /* Set new frame pointer                  */
        reg.sp := reg.sp - reg.inst_reg.ab;        /* Allocate for automatics                */
        reg.index.pc := reg.index.eb + reg.scr1 + 1;/* First instruction of proc             */

        mm_index_select := n_pc;
        call MAIN_MEM.READ_I;                      /* read onto bus for decode               */

        do   act MAIN_MEM.READ_P1                  /* Activate instr. pipeline               */
         []  act INSTRUCTION_DECODE.SELECT_NEXT
        od;

    endproc;

endmech;

mech  DEPOSIT_OPRND;
    /*
     *    Context of activation :  By instruction execution to write destination
     *                             operand back to memory.
     *
     */

    Not shown ....

endmech;
```

```
      endsys;

endsys;

sys  MEMORY;
     /*
      *  Memory subsystem controls the access to the two
      *  QM-C memories - Main memory and auxillary memory.
      *
      *  Subsystems   : None
      *
      *  Mechanisms   : MAIN_MEM
      *                 AUX_MEM
      *
      */

      Not shown ...

endsys;

sys INTERRUPT ;
    /*  Describes the interrupt mechanisms.
     *
     *  Subsystems   :  None.
     *
     *  Mechanisms   :  INT_HANDLERS
     *                  CHANGE_STATE
     *
     */

   Not shown....

endsys;

sys NANO_ARCHITECTURE;

     /*
      *  Contains mechanisms, not described fruther at this level.
      *  These mechanisms are QM-1 functions, whose internal function
      *  is not of interest for the QM-C architecture.
      *
      *  Subsystems :  None.
      *
      *  Mechanisms :  EXEC_PRIM  -  Nanoword execution matrix.
      *                              EXEC_NPC executes nanoword pointed to by
      *                                       address in "inst_addr".
      *                              EXEC_ALU performs alu operation in "alu_operation"
      *                                       on registers pointed to by "source"
```

```
                *                                       and "alt_source".
                *
                *               IO_PRIM     -  Lowest level input/output mechanism.
                *                              READ_BUFF_ADDR reads buffer address from channel
                *                                             to port[ port_num ].
                *                              READ_DATA reads data from channel to port
                *                              TRANSMIT_DATA writes data from port to channel
                *
                *               MEM_PRIM    -  Main store accessing primitives.
                *                              MS_INITIATE starts ms access cycle.
                *
                */

            mech  EXEC_PRIM;

                proc  EXEC_NPC; ... endproc;
                proc  EXEC_ALU; ... endproc;

            endmech;

            mech   IO_PRIM;

                proc READ_BUFF_ADDR; ... endproc;
                proc READ_DATA;      ... endproc;
                proc TRANSMIT_DATA;  ... endproc;

            endmech;

            mech   MEM_PRIM;

                proc  MS_INITIATE;  ... endproc;

            endmech;

        endsys;

endsys ;
```

Appendix IV

Excerpts from the S*(QM-1) code for the QM-C instruction set

```
prog( qmc )

    declaration

        .
        .
        Declaration of QM-1 fixed resources
        .
        .
        QM-C resources declared as synonyms to QM-1 fixed resources
        .
        .
        Macro declarations :
            FETCH_OPRND macros        -  Correspond to the FETCH_OPRND mechanism in S*A
            MEMORY macros             -  Correspond to the MEMORY mechanism
            INSTRUCTION_FETCH macros  -  Correspond to portions of the INSTRUCTION_FETCH
                                         mechanism
        .
        .
    enddec

    proc INCM (instruction)  ... endproc
        .
        .
    proc p_call (instruction) ... endproc
        .
        .
    interrupt processes    - Correspond to the INTERRUPT system in S*A
    subroutine processes   - Processes not implemented as macros, for example
                             the DEPOSIT_OPRND mechanism
        .
        .
endprog
```

```
/*
 *  The following is an exerpt from the QM-C macro declarations
 *
 */

macro FETCH_OPRND_FORM_2

  /*
   *                    ((r)+o18) <- ((r)+o18)  op  c6
   *
   *  Precondition                                  Postcondition :
   *    r in inst_reg.ap                             ((r)+o18) in reg[ dest ]
   *    c6 in inst_reg.bp                              (r)+o18 in reg[ addr ]
   *    o18 on main_output bus                             c6  in reg[ source ]
   *
   */

    INC_PROG_COUNT;                          /* Correct for 2-w instruction  */

    scr2 := main_output;                     /* Calc. address of destination */
    source := inst_reg.ap;
    scr2 := scr2 + reg[ source ];

    inst_reg.c := 0;                         /* Get source                   */
    scr1 := xpass1 inst_reg;

    addr := c_scr2;                          /* Get destination              */
    MAIN_MEM_READ_G;
    inst_reg := main_output;                 /* Inst. reg stores destination */

    dest       := c_inst_reg;                /* Setup for operation          */
    source     := c_scr1;
    alt_source := c_inst_reg

endmacro

macro MAIN_MEM_READ_I
        /*
         *  Read main memory     - Address from fp, eb or pc. "mm_index_select"
         *                         must be set up by caller to select any of these.
         *                         Data appears on the MM output bus.
         *
         *
         */

       main_output := main_memory[ mm_addr_so.index[ mm_index_select ] ]

endmacro

macro MAIN_MEM_READ_I_L
```

```
        /*
         *  Read main memory     - Address formed by adding the 11 bit "b" field
         *                         of the instruction register, sign extended to
         *                         the index register (fp, ed or pc) selected by
         *                         "mm_index_select". Data appears on MM output bus.
         *
         */

        main_output := main_memory[ mm_addr_so.index[ mm_index_select ] +ab ]

endmacro

macro MAIN_MEM_PUSH_G
        /*
         *  Write main memory     - Address in general register. "mm_addr_select"
         *                          must be set up by caller to select register.
         *                          "mm_data_select" selects data source.
         *                          Register decremented by 1 after write.
         *
         *     Note that data written appears on the MM output bus.
         *
         */

        main_memory[ mm_addr_so.reg[ mm_addr_select ] ] := mm_data_so[ mm_data_select ];
        ks := mm_addr_select;
        reg[ gspec.ks ] := xdecl reg[ gspec.ks ]

endmacro

macro INSTRUCTION_FETCH_NEXT
    /*
     *  Context of Activation :  At end of each instruction and internal
     *                           interrupt handlers.
     *
     *     As instruction i activates FETCH, decoding of instruction i+1
     *     is initiated concurrent with fetch of instruction i+2.
     *
     *     This macro assumes program counter already set.
     *
     */

    load_npc := c_mem;              /* Initatie decode/select next cycle  */
    load_r31 := 1;                  /* Load inst_reg from main_output bus */

    main_output := main_mem[ reg.index[ mm_index_select ] +1 ]

endmacro

macro  INC_PROG_COUNT
```

```
    */
     *    Context of activation :  Instructions and interrupt handlers to
     *                             increment program counter by 1.
     *
     *              Assumes mm_index_select points to program counter.
     *
     */

    reg.index[ mm_index_select ] := reg.index[ mm_index_select ] +1

endmacro

macro  SET_STATUS
    /*
     *    Context of activation :  Arithmetic instructions and compare to
     *                             set global condition codes.
     *
     */

    alu_status   := 1;
    shift_status := 1

endmacro
```

```
/*
 *  Two instruction processes shown as an example of the S*(QM-1)
 *  code. These processes use the macros shown above.
 *
 */

proc INCM (instruction)

    /*
     *  Increment memory
     *
     *    Format   :  incm    r, c6, o18
     *
     *    Function :  ((r)+o18) <- ((r)+o18) + c6
     *
     */

    FETCH_OPRND_FORM_2;                          /* Fetch operands          */

    reg[ dest ] := reg[ alt_source ] + reg[ source ];
    SET_STATUS;

    act  write_reg:                              /* Write to mem. and fetch next */

endproc

proc p_call (instruction, mnemonic=call)

    /*
     *  Procedure call.  First word of each procedure is a mask word indicating
     *                   the number of registers used (thus have to be saved)
     *                   and size of stack to be allocated for automatics and
     *                   temporaries.
     *
     *  Format   :  call  o11
     *
     *  Function : save registers 'pc' through register indicated in mask,
     *             allocate automatic storage on stack and transfer control
     *             to called procedure
     *
     */

    INC_PROG_COUNT;                          /* Point pc to next instruction */
    scr1 := passl inst_reg;                  /* Save offset                  */

    mm_index_select := c_eb;                 /* Read from external base      */
    mm_addr_select := c_sp;                  /* Save on stack                */
    mm_data_select := c_pc_p1;               /* PC first saved (c_pc+1)      */
```

```
MAIN_MEM_READ_I_L;                    /* Read mask word                */
inst_reg := main_output;              /* Prepare to decode             */


repeat                                /* Save registers                */
    mm_data_select := mm_data_select -1;

    main_memory[mm_addr_so.reg[mm_addr_select]] := mm_data_so[mm_data_select];
    ks := c_sp;
    reg[ gspec.ks ] := xdecl reg[ gspec.ks ]

    f_scri := mm_data_select - inst_reg.c
until  (f_scri == 0);

inst_reg.c := 0;                      /* Prep. to allocate automatics */
fp := passl sp;                       /* Set new frame pointer        */
sp := sp - inst_reg;                  /* Allocate for automatics      */

alu_carry_in := 1;                    /* Add one to following expr.   */
pc := eb + scri;                      /* First instruction of proc    */

mm_index_select := c_pc;
MAIN_MEM_READ_I;                      /* Read onto bus for decode     */

INSTRUCTION_FETCH_NEXT;               /* Activate instr. pipeline     */

endproc
```

```
/*
 *  An example of an S*(QM-1) subroutine procedure. Activated by
 *  the INCM procedure shown above
 *
 */

proc write_reg (subroutine)
    /*
     *  Write destination operand to memory and initiate instruction
     *  prefetch. Destination format : reg + offset.
     *
     *    Precondition                        Postcondition
     *      Value of destination            (main_memory[ (reg[ addr ]) ] = (reg[ dest ])
     *      in reg[ dest ].                 (main_memory[ (pc)+1 ]) being decoded.
     *      Address of destination          (main_memory[ (pc)+2 ]) being fetched.
     *      in reg[ addr ].
     *
     */

    mm_data_so := dest;                     /* Set up source for write */
    MAIN_MEM_WRITE_G;                       /* Write to memory  */

    MAIN_MEM_READ_P1;                       /* Read next instruction */
    INSTRUCTION_FETCH_NEXT_P;               /* Start instruction pipeline */

endproc
```